

Freescale Semiconductor, Inc.

**CodeWarrior™  
Development Studio for  
Freescale™ 56800/E  
Hybrid Controllers:  
MC56F83xx/DSP5685x  
Family  
Targeting Manual**

Revised 28 October 2004

**metrowerks**

For More Information: [www.freescale.com](http://www.freescale.com)

# Freescale Semiconductor, Inc.

Metrowerks and the Metrowerks logo are registered trademarks of Metrowerks Corporation in the United States and/or other countries. CodeWarrior is a trademark or registered trademark of Metrowerks Corporation in the United States and/or other countries. All other trade names and trademarks are the property of their respective owners.

Copyright © 2004 Metrowerks Corporation. ALL RIGHTS RESERVED.

**No portion of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, without prior written permission from Metrowerks. Use of this document and related materials are governed by the license agreement that accompanied the product to which this manual pertains. This document may be printed for non-commercial personal use only in accordance with the aforementioned license agreement. If you do not have a copy of the license agreement, contact your Metrowerks representative or call 1-800-377-5416 (if outside the U.S., call +1-512-996-5300).**

Metrowerks reserves the right to make changes to any product described or referred to in this document without further notice. Metrowerks makes no warranty, representation or guarantee regarding the merchantability or fitness of its products for any particular purpose, nor does Metrowerks assume any liability arising out of the application or use of any product described herein and specifically disclaims any and all liability. **Metrowerks software is not authorized for and has not been designed, tested, manufactured, or intended for use in developing applications where the failure, malfunction, or any inaccuracy of the application carries a risk of death, serious bodily injury, or damage to tangible property, including, but not limited to, use in factory control systems, medical devices or facilities, nuclear facilities, aircraft navigation or communication, emergency systems, or other applications with a similar degree of potential hazard.**

## How to Contact Metrowerks

Corporate Headquarters	Metrowerks Corporation 7700 West Parmer Lane Austin, TX 78729 U.S.A.
World Wide Web	<a href="http://www.metrowerks.com">http://www.metrowerks.com</a>
Sales	United States Voice: 800-377-5416 United States Fax: 512-996-4910 International Voice: +1-512-996-5300 Email: <a href="mailto:sales@metrowerks.com">sales@metrowerks.com</a>
Technical Support	United States Voice: 800-377-5416 International Voice: +1-512-996-5300 Email: <a href="mailto:support@metrowerks.com">support@metrowerks.com</a>

# Table of Contents

---

<b>1</b>	<b>Introduction</b>	<b>13</b>
	CodeWarrior IDE . . . . .	13
	Freescale 56800/E Hybrid Controllers . . . . .	15
	References. . . . .	16
<b>2</b>	<b>Getting Started</b>	<b>19</b>
	System Requirements . . . . .	19
	Installing and Registering the CodeWarrior IDE . . . . .	20
	Creating a Project . . . . .	23
<b>3</b>	<b>Development Studio Overview</b>	<b>35</b>
	CodeWarrior IDE . . . . .	35
	Development Process . . . . .	36
	Project Files . . . . .	38
	Editing Code . . . . .	39
	Building: Compiling and Linking . . . . .	40
	Debugging . . . . .	42
<b>4</b>	<b>Target Settings</b>	<b>43</b>
	Target Settings Overview . . . . .	43
	Target Setting Panels . . . . .	43
	Changing Target Settings . . . . .	45
	Exporting and Importing Panel Options to XML Files . . . . .	47
	Restoring Target Settings . . . . .	47
	CodeWarrior IDE Target Settings Panels . . . . .	48
	DSP56800E-Specific Target Settings Panels . . . . .	48
	Target Settings . . . . .	49
	M56800E Target . . . . .	50
	C/C++ Language (C Only) . . . . .	51

# Freescale Semiconductor, Inc.

## Table of Contents

---

C/C++ Preprocessor . . . . .	54
C/C++ Warnings . . . . .	56
M56800E Assembler . . . . .	60
M56800E Processor . . . . .	62
ELF Disassembler . . . . .	65
M56800E Linker . . . . .	67
Remote Debugging . . . . .	72
M56800E Target (Debugging) . . . . .	73
Remote Debug Options . . . . .	77
<b>5 Processor Expert Interface</b>	<b>81</b>
Processor Expert Overview . . . . .	81
Processor Expert Code Generation . . . . .	82
Processor Expert Beans . . . . .	83
Processor Expert Menu . . . . .	85
Processor Expert Windows . . . . .	89
Bean Selector . . . . .	89
Bean Inspector . . . . .	90
Target CPU Window . . . . .	92
Memory Map Window . . . . .	97
CPU Types Overview . . . . .	98
Resource Meter . . . . .	99
Installed Beans Overview . . . . .	100
Peripherals Usage Inspector . . . . .	101
Processor Expert Tutorial . . . . .	102
<b>6 C for DSP56800E</b>	<b>119</b>
Number Formats . . . . .	120
Calling Conventions and Stack Frames . . . . .	121
Passing Values to Functions . . . . .	121
Returning Values From Functions . . . . .	122
Volatile and Non-Volatile Registers . . . . .	122
Stack Frame and Alignment . . . . .	125

---



# Freescale Semiconductor, Inc.

## Table of Contents

User Stack Allocation . . . . .	126
Data Alignment Requirements . . . . .	131
Word and Byte Pointers . . . . .	132
Reordering Data for Optimal Usage . . . . .	132
Variables in Program Memory . . . . .	133
Declaring Program Memory Variables . . . . .	133
Using Variables in Program Memory . . . . .	134
Linking with Variables in Program Memory . . . . .	136
Packed Structures Support . . . . .	138
Code and Data Storage . . . . .	139
Large Data Model Support . . . . .	141
Extended Data Addressing Example . . . . .	142
Accessing Data Objects Examples . . . . .	142
External Library Compatibility . . . . .	144
Optimizing Code . . . . .	144
Deadstripping and Link Order . . . . .	145
Working with Peripheral Module Registers . . . . .	146
Compiler Generates Bit Instructions . . . . .	146
Explanation of Undesired Behavior . . . . .	148
Recommended Programming Style . . . . .	149
Generating MAC Instruction Set . . . . .	150
<b>7 High-Speed Simultaneous Transfer</b>	<b>153</b>
Host-Side Client Interface. . . . .	153
hsst_open . . . . .	153
hsst_close . . . . .	154
hsst_read . . . . .	155
hsst_write . . . . .	156
hsst_size . . . . .	157
hsst_block_mode . . . . .	157
hsst_noblock_mode . . . . .	158
hsst_attach_listener . . . . .	158
hsst_detach_listener . . . . .	159

# Freescale Semiconductor, Inc.

## Table of Contents

---

hsst_set_log_dir . . . . .	160
HSST Host Program Example . . . . .	160
Target Library Interface . . . . .	162
HSST_open . . . . .	162
HSST_close . . . . .	162
HSST_setvbuf . . . . .	163
HSST_write . . . . .	164
HSST_read . . . . .	165
HSST_flush . . . . .	166
HSST_size . . . . .	166
HSST_raw_read . . . . .	167
HSST_raw_write . . . . .	167
HSST_set_log_dir . . . . .	168
HSST Target Program Example . . . . .	169
<b>8 Data Visualization</b>	<b>171</b>
Starting Data Visualization . . . . .	171
Data Target Dialog Boxes . . . . .	173
Memory . . . . .	173
Registers . . . . .	174
Variables . . . . .	175
HSST . . . . .	176
Graph Window Properties . . . . .	177
<b>9 Debugging for DSP56800E</b>	<b>179</b>
Target Settings for Debugging . . . . .	179
Command Converter Server . . . . .	180
Essential Target Settings for Command Converter Server . . . . .	181
Changing the Command Converter Server Protocol to Parallel Port . . . . .	181
Changing the Command Converter Server Protocol to HTI . . . . .	184
Changing the Command Converter Server Protocol to PCI . . . . .	184
Setting Up a Remote Connection . . . . .	185
Debugging a Remote Target Board . . . . .	188

---

# Freescale Semiconductor, Inc.

## Table of Contents

Launching and Operating the Debugger . . . . .	188
Setting Breakpoints and Watchpoints . . . . .	192
Viewing and Editing Register Values . . . . .	193
Viewing X: Memory . . . . .	195
Viewing P: Memory . . . . .	197
Load/Save Memory . . . . .	200
Fill Memory . . . . .	202
Save/Restore Registers . . . . .	204
EOnCE Debugger Features . . . . .	206
Set Hardware Breakpoint Panel . . . . .	207
Special Counters . . . . .	208
Trace Buffer . . . . .	209
Set Trigger Panel . . . . .	212
Using the DSP56800E Simulator . . . . .	214
Cycle/Instruction Count . . . . .	215
Memory Map . . . . .	216
Register Details Window . . . . .	216
Loading a .elf File without a Project. . . . .	217
Using the Command Window . . . . .	218
System-Level Connect . . . . .	219
Debugging in the Flash Memory . . . . .	219
Flash Memory Commands . . . . .	220
set_hfmcld <value> . . . . .	220
set_hfm_base <address> . . . . .	220
set_hfm_config_base <address> . . . . .	221
add_hfm_unit <startAddr> <endAddr> <bank> <numSectors> <pageSize> <progMem> <boot> <interleaved> . . . . .	221
set_hfm_erase_mode units   pages   all . . . . .	221
set_hfm_verify_erase 1   0 . . . . .	222
set_hfm_verify_program 1   0 . . . . .	222
target_code_sets_hfmcld 1   0 . . . . .	222
Flash Lock/Unlock . . . . .	222
Notes for Debugging on Hardware . . . . .	223

# Freescale Semiconductor, Inc.

## Table of Contents

---

<b>10 Profiler</b>	<b>225</b>
<b>11 Inline Assembly Language and Intrinsics</b>	<b>227</b>
Inline Assembly Language . . . . .	227
Inline Assembly Overview . . . . .	228
Assembly Language Quick Guide . . . . .	229
Calling Assembly Language Functions from C Code . . . . .	230
Calling Functions from Assembly Language . . . . .	232
Intrinsic Functions . . . . .	233
Implementation . . . . .	233
Fractional Arithmetic . . . . .	234
Intrinsic Functions for Math Support . . . . .	235
abs_s . . . . .	237
negate . . . . .	237
L_abs . . . . .	238
L_negate . . . . .	238
add . . . . .	239
sub . . . . .	240
L_add . . . . .	241
L_sub . . . . .	241
stop . . . . .	242
wait . . . . .	243
turn_off_conv_rndg . . . . .	243
turn_off_sat . . . . .	243
turn_on_conv_rndg . . . . .	244
turn_on_sat . . . . .	244
extract_h . . . . .	245
extract_l . . . . .	245
L_deposit_h . . . . .	246
L_deposit_l . . . . .	246
div_s . . . . .	247
div_s4q . . . . .	248
div_ls . . . . .	248

---

# Freescale Semiconductor, Inc.

## Table of Contents

div_ls4q . . . . .	249
mac_r . . . . .	250
msu_r . . . . .	251
mult . . . . .	252
mult_r . . . . .	252
L_mac . . . . .	253
L_msu . . . . .	254
L_mult . . . . .	254
L_mult_ls . . . . .	255
ffs_s . . . . .	256
norm_s . . . . .	256
ffs_l . . . . .	257
norm_l . . . . .	258
round . . . . .	259
shl . . . . .	260
shlftNs . . . . .	261
shlfts . . . . .	262
shr . . . . .	263
shr_r . . . . .	264
shrtNs . . . . .	264
L_shl . . . . .	265
L_shlftNs . . . . .	266
L_shlfts . . . . .	267
L_shr . . . . .	267
L_shr_r . . . . .	268
L_shrtNs . . . . .	269
Modulo Addressing Intrinsic Functions . . . . .	269
__mod_init . . . . .	271
__mod_initint16 . . . . .	272
__mod_start . . . . .	272
__mod_access . . . . .	273
__mod_update . . . . .	273
__mod_stop . . . . .	273

## Table of Contents

---

__mod_getint16 . . . . .	274
__mod_setint16 . . . . .	274
__mod_error . . . . .	275
 <b>12 ELF Linker</b>	 <b>281</b>
Structure of Linker Command Files . . . . .	281
Memory Segment . . . . .	281
Closure Blocks . . . . .	282
Sections Segment . . . . .	283
Linker Command File Syntax . . . . .	284
Alignment . . . . .	284
Arithmetic Operations . . . . .	285
Comments . . . . .	285
Deadstrip Prevention . . . . .	286
Variables, Expressions, and Integral Types . . . . .	286
File Selection . . . . .	288
Function Selection . . . . .	289
ROM to RAM Copying . . . . .	289
Utilizing Program Flash and Data RAM for Constant Data in C . . . . .	291
Utilizing Program Flash for User-Defined Constant Section in Assembler . . . . .	292
Stack and Heap . . . . .	294
Writing Data Directly to Memory . . . . .	294
Linker Command File Keyword Listing . . . . .	294
. (location counter) . . . . .	294
ADDR . . . . .	295
ALIGN . . . . .	296
ALIGNALL . . . . .	296
FORCE_ACTIVE . . . . .	297
INCLUDE . . . . .	297
KEEP_SECTION . . . . .	298
MEMORY . . . . .	298
OBJECT . . . . .	300
REF_INCLUDE . . . . .	300

---

SECTIONS . . . . .	300
SIZEOF . . . . .	302
SIZEOFW . . . . .	303
WRITEB . . . . .	303
WRITEH . . . . .	303
WRITEW . . . . .	304
<b>13 Command-Line Tools</b>	<b>305</b>
Usage . . . . .	305
Response File . . . . .	306
Sample Build Script . . . . .	307
Arguments. . . . .	307
General Command-Line Options . . . . .	307
Compiler . . . . .	309
Linker . . . . .	316
Assembler . . . . .	320
<b>14 Libraries and Runtime Code</b>	<b>321</b>
MSL for DSP56800E . . . . .	321
Using MSL for DSP56800E . . . . .	321
Allocating Stacks and Heaps for the DSP56800E . . . . .	324
Runtime Initialization . . . . .	325
EOnCE Library . . . . .	328
_eonce_Initialize . . . . .	330
_eonce_SetTrigger . . . . .	331
_eonce_SetCounterTrigger . . . . .	332
_eonce_ClearTrigger . . . . .	333
_eonce_GetCounters . . . . .	334
_eonce_GetCounterStatus . . . . .	334
_eonce_SetupTraceBuffer . . . . .	335
_eonce_GetTraceBuffer . . . . .	336
_eonce_ClearTraceBuffer . . . . .	336
_eonce_StartTraceBuffer . . . . .	337

# Freescale Semiconductor, Inc.

## Table of Contents

---

_eonce_HaltTraceBuffer . . . . .	337
_eonce_EnableDEBUGEV . . . . .	338
_eonce_EnableLimitTrigger . . . . .	338
Definitions . . . . .	339
<b>A Porting Issues</b>	<b>349</b>
Converting the DSP56800E Projects from Previous Versions . . . . .	349
Removing "illegal object_c on pragma directive" Warning . . . . .	350
<b>B DSP56800x New Project Wizard</b>	<b>351</b>
Overview . . . . .	351
Page Rules . . . . .	353
Resulting Target Rules . . . . .	356
Rule Notes . . . . .	356
DSP56800x New Project Wizard Graphical User Interface . . . . .	357
Invoking the New Project Wizard . . . . .	357
New Project Dialog Box . . . . .	358
Target Pages . . . . .	359
Program Choice Page . . . . .	363
Data Memory Model Page . . . . .	365
External/Internal Memory Page . . . . .	366
Finish Page . . . . .	367



# Introduction

---

This manual explains how to use the CodeWarrior™ Integrated Development Environment (IDE) to develop code for the DSP56800E family of processors (MC56F3xx and DSP56F5x).

This chapter contains the following sections:

- CodeWarrior IDE
- Freescale 56800/E Hybrid Controllers
- References

## CodeWarrior IDE

The CodeWarrior IDE consists of a project manager, a graphical user interface, compilers, linkers, a debugger, a source-code browser, and editing tools. You can edit, navigate, examine, compile, link, and debug code, within the one CodeWarrior environment. The CodeWarrior IDE lets you configure options for code generation, debugging, and navigation of your project.

Unlike command-line development tools, the CodeWarrior IDE organizes all files related to your project. You can see your project at a glance, so organization of your source-code files is easy. Navigation among those files is easy, too.

When you use the CodeWarrior IDE, there is no need for complicated build scripts of makefiles. To add files to your project or delete files from your project, you use your mouse and keyboard, instead of tediously editing a build script.

For any project, you can create and manage several configurations for use on different computer platforms. The platform on which you run the CodeWarrior IDE is called the host. From the host, you use the CodeWarrior IDE to develop code to target various platforms.

Note the two meanings of the term *target*:

- **Platform Target** — The operating system, processor, or microcontroller for which/on which your code will execute.

# Freescale Semiconductor, Inc.

## Introduction *CodeWarrior IDE*

---

- **Build Target** — The group of settings and files that determine what your code is, as well as control the process of compiling and linking.

The CodeWarrior IDE lets you specify multiple build targets. For example, a project can contain one build target for debugging and another build target optimized for a particular operating system (platform target). These build targets can share files, even though each build target uses its own settings. After you debug the program, the only actions necessary to generate a final version are selecting the project's optimized build target and using a single Make command.

The CodeWarrior IDE's extensible architecture uses plug-in compilers and linkers to target various operating systems and microprocessors. For example, the IDE uses a GNU tool adapter for internal calls to DSP56800E development tools.

Most features of the CodeWarrior IDE apply to several hosts, languages, and build targets. However, each build target has its own unique features. This manual explains the features unique to the CodeWarrior Development Studio for Freescale 56800/E Hybrid Controllers.

For comprehensive information about the CodeWarrior IDE, see the *CodeWarrior IDE User's Guide*.

---

<b>NOTE</b>	For the very latest information on features, fixes, and other matters, see the <i>CodeWarrior Release Notes</i> , on the CodeWarrior IDE CD.
-------------	--

---

## Freescale 56800/E Hybrid Controllers

The Freescale 56800/E Hybrid Controllers consist of two sub-families, which are named the DSP56F80x/DSP56F82x (DSP56800) and the MC56F83xx/DSP5685x (DSP56800E). The DSP56800E is an enhanced version of the DSP56800.

The processors in the the DSP56800 and DSP56800E sub-families are shown in Table 1.1.

With this product the following Targeting Manuals are included:

- *Code Warrior Development Studio for Freescale 56800/E Hybrid Controllers: DSP56F80x/DSP56F82x Family Targeting Manual*
- *Code Warrior Development Studio for Freescale 56800/E Hybrid Controllers: MC56F83xx/DSP5685x Family Targeting Manual*

---

**NOTE** Please refer to the Targeting Manual specific to your processor.

---

**Table 1.1 Supported DSP56800x Processors for CodeWarrior Development Studio for Freescale 56800/E Hybrid Controllers**

DSP56800	DSP56800E
DSP56F801 (60 MHz)	DSP56852
DSP56F801 (80 MHz)	DSP56853
DSP56F802	DSP56854
DSP56F803	DSP56855
DSP56F805	DSP56857
DSP56F807	DSP56858
DSP56F826	MC56F8322
DSP56F827	MC56F8323
	MC56F8345
	MC56F8346
	MC56F8356
	MC56F8357
	MC56F8365
	MC56F8366

**Table 1.1 Supported DSP56800x Processors for CodeWarrior Development Studio for Freescale 56800/E Hybrid Controllers (cont.)**

DSP56800	DSP56800E
	MC56F8367
	MC56F8122
	MC56F8123
	MC56F8145
	MC56F8146
	MC56F8147
	MC56F8155
	MC56F8156
	MC56F8157
	MC56F8165
	MC56F8166
	MC56F8167

## References

- Your CodeWarrior IDE includes these manuals:
  - *Code Warrior IDE User's Guide*
  - *CodeWarrior Development Studio IDE 5.6 Windows® Automation Guide*
  - *Code Warrior Development Studio for Freescale 56800/E Hybrid Controllers: DSP56F80x/DSP56F82x Family Targeting Manual*
  - *Code Warrior Development Studio for Freescale 56800/E Hybrid Controllers: MC56F83xx/DSP5685x Family Targeting Manual*
  - *Code Warrior Builds Tools Reference for Freescale 56800/E Hybrid Controllers*
  - *Code Warrior Development Studio IDE 5.5 User's Guide Profiler Supplement*
  - *Code Warrior Development Studio HTI HostTarget Interface (for Once™/JTAG Communication) User's Manual*
  - *Assembler Reference Manual*
  - *MSL C Reference* (Metrowerks Standard C libraries)

- *DSP56800 to DSP56800E Porting Guide*. Freescale Semiconductors, Inc.
- To learn more about the DSP56800E processor, refer to the Freescale manual, *DSP56800E Family Manual*.

To download electronic copies of these manuals or order printed versions, visit:

<http://www.freescale.com/>

# Freescale Semiconductor, Inc.

**Introduction**  
*References*

---

# Getting Started

---

This chapter explains the setup and installation for the CodeWarrior™ IDE, including hardware connections and communications protocols.

This chapter contains these sections:

- System Requirements
- Installing and Registering the CodeWarrior IDE
- Creating a Project

## System Requirements

Table 2.1 lists system requirements for installing and using the CodeWarrior IDE for DSP56800E.

**Table 2.1 Requirements for the CodeWarrior IDE**

Category	Requirement
Host Computer Hardware	PC or compatible host computer with 133-megahertz Pentium®-compatible processor, 64 megabytes of RAM, and a CD-ROM drive
Operating System	Microsoft® Windows® 98/2000/NT/XP
Hard Drive	1.2 gigabytes of free space, plus space for user projects and source code
DSP56800E	56800E EVM or custom 56800E development board, with JTAG header
Other	Power supply

## Getting Started

Installing and Registering the CodeWarrior IDE

---

# Installing and Registering the CodeWarrior IDE

Follow these steps:

1. To install the CodeWarrior software:
  - a. Insert the CodeWarrior CD into the CD-ROM drive — the welcome screen appears.

---

<b>NOTE</b>	If the Auto Install is disabled, run the program <code>Setup.exe</code> in the root directory of the CD.
-------------	--

---

- b. Click **Launch CodeWarrior Setup** — the install wizard displays welcome page.
  - c. Follow the wizard instructions, accepting all the default settings.
  - d. At the prompt to check for updates, click the **Yes** button — the CodeWarrior updater opens.
2. To check for updates:

---

<b>NOTE</b>	If the updater already has Internet connection settings, you may proceed directly to substep f.
-------------	---

---

- a. Click the **Settings** button — the **Updater Settings** dialog box appears.
  - b. Click the **Load Settings** button — the updater loads settings from your Windows control panel.
  - c. Modify the settings, as appropriate.
  - d. If necessary, enter the proxy username and the password.
  - e. Click the **Save** button — the **Updater Settings** dialog box disappears.
  - f. In the updater screen, click the **Check for Updates** button.
  - g. If updates are available, follow the on-screen instructions to download the updates to your computer.
  - h. When you see the message, “Your version ... is up to date”, click the **OK** button — the message box closes.



- i. Click the updater **Close** button — the installation resumes.
  - j. At the prompt to restart the computer, select the **Yes** option button.
  - k. Click the **Finish** button — the computer restarts, completing installation.
3. To register the CodeWarrior software:
  - a. Select **Start> Programs>Metrowerks CodeWarrior>CodeWarrior for Freescale 56800/E Hybrid Controllers R7.0>CodeWarrior IDE** — the registration window appears.

**Figure 2.1 CodeWarrior Registration Window**



---

**NOTE** To evaluate this product before purchasing it, click **Register Later** and skip to Step 4.

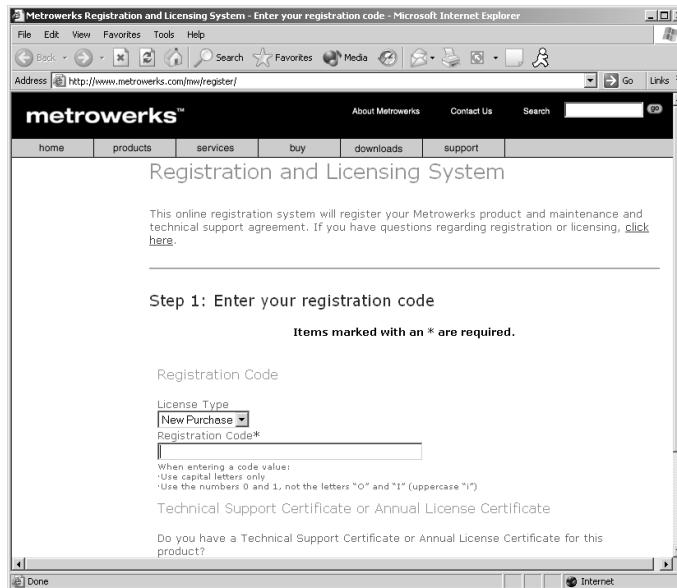
---

- b. Click **Register Now** — the Metrowerks registration web page appears in the web browser window.

## Getting Started

### Installing and Registering the CodeWarrior IDE

**Figure 2.2** Metrowerks Registration Web Page



- c. Follow the instructions to complete the registration — Metrowerks will send you the license authorization code to your e-mail address.
- d. Close the web browser window.
- e. Check your e-mail and retrieve the license authorization code.

---

**NOTE** If you encounter difficulty during the registration process, send an e-mail to [license@metrowerks.com](mailto:license@metrowerks.com).

---

- f. From the CodeWarrior menu bar, select **Help > License Authorization** — the **License Authorization** dialog box appears.
- g. Enter the license authorization code that Metrowerks sent you.

---

**NOTE** To avoid transcription errors, we recommend that you copy and paste the license authorization code rather than typing it in. Metrowerks license authorization codes do not contain the letters O or I.

---

- h. Click the **OK** button — this completes the installation and registration.

4. Your CodeWarrior software is ready for use.
  - a. Table 2.2 lists the directories created during full installation.
  - b. To test your system, follow the instructions of the next section to create a project.

**Table 2.2 Installation Directories, CodeWarrior IDE for DSP56800E**

Directory	Contents
(CodeWarrior_Examples)	Target-specific projects and code.
(Helper Apps)	Applications such as cwspawn.exe and cvs.exe.
bin	The CodeWarrior IDE application and associated plug-in tools.
ccs	Command converter server executable files and related support files.
DSP56800x_EABI_Support	Default files for the DSP56800x stationery.
DSP56800x_EABI_Tools	Drivers to the CCS and command line tools, plus IDE default files for the DSP56800x stationery
Freescal_Documentation	Documentation specific to the Freescal DSP56800E series.
Help	Core IDE and target-specific help files. (Access help files through the Help menu or F1 key.)
License	The registration program and additional licensing information.
M56800E Support	Initialization files, Metrowerks Standard Library (MSL) and Runtime Library.
M56800x Support	Profiler libraries.
Other_Metrowerks_Tool	MWRremote executable file.
ProcessorExpert	Files for the Processor Expert.
Release_Notes	Release notes for the CodeWarrior IDE and each tool.
Stationery	Templates for creating DSP56800E projects. Each template pertains to a specific debugging protocol.

## Creating a Project

To test software installation, create a sample project. Follow these steps:

## Getting Started

### Creating a Project

---

1. Select **Start>Metrowerks CodeWarrior>CW for DSP56800 R7.0>CodeWarrior IDE**. The IDE starts; the main window appears.

To create a DSP56800x project use either the:

- DSP56800x new project wizard
- DSP56800x EABI stationery

To create a new project with the DSP56800x new project wizard, please see the subsection “Creating a New Project with the DSP56800x New Project Wizard.”

To create a new project with the DSP56800x EABI stationery, please see the subsection “Creating a New Project with the DSP56800x EABI Stationery.”

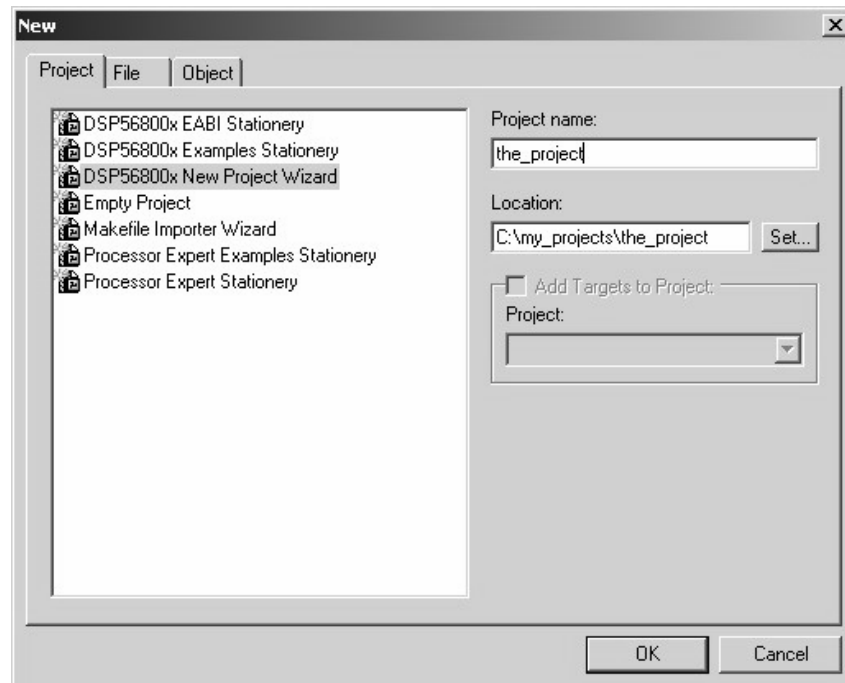
## Creating a New Project with the DSP56800x New Project Wizard

In this section of the tutorial, you work with the CodeWarrior IDE to create a project with the DSP56800x New Project Wizard.

To create a project:

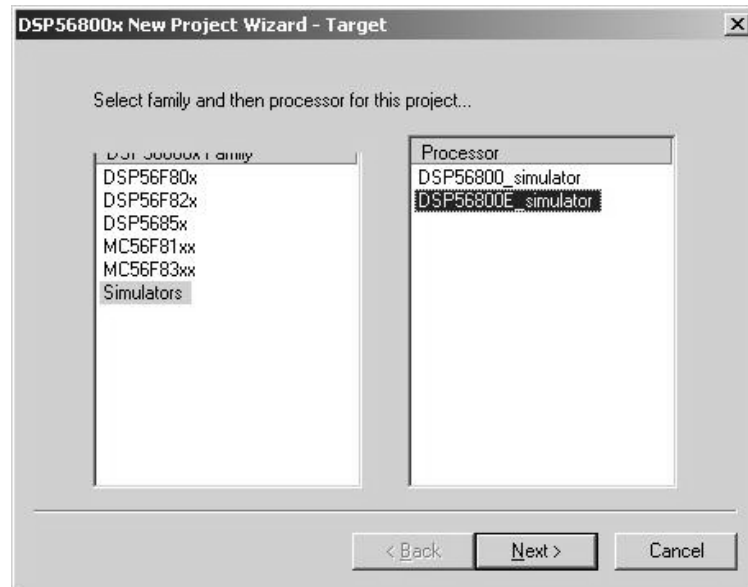
1. From the menu bar of the Metrowerks CodeWarrior window, select **File>New**.  
The **New** dialog box (Figure 2.3) appears.

**Figure 2.3 New Dialog Box**



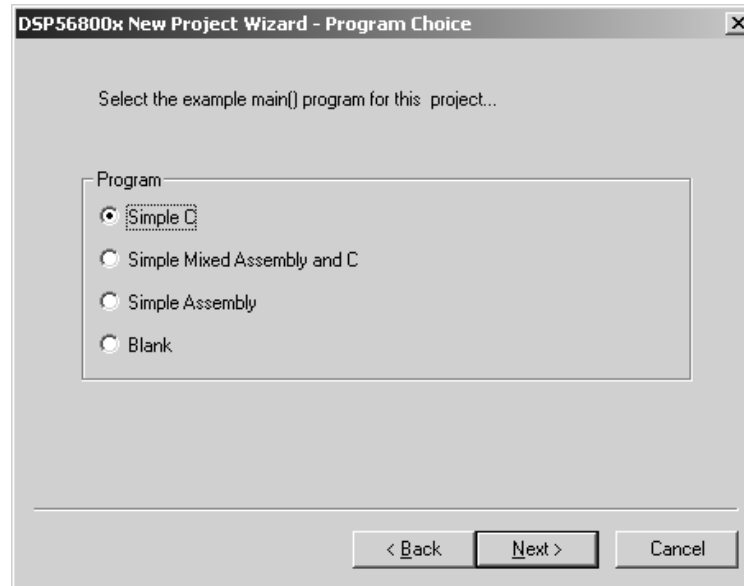
2. Select **DSP56800x New Project Wizard**.
3. In the **Project Name** text box, type the project name. For example, the\_project.
4. In the **Location** text box, type the location where you want to save this project or choose the default location.
5. Click **OK**. The **DSP56800x New Project Wizard — Target** dialog box (Figure 2.4) appears.

**Figure 2.4 DSP56800x New Project Wizard — Target Dialog Box**



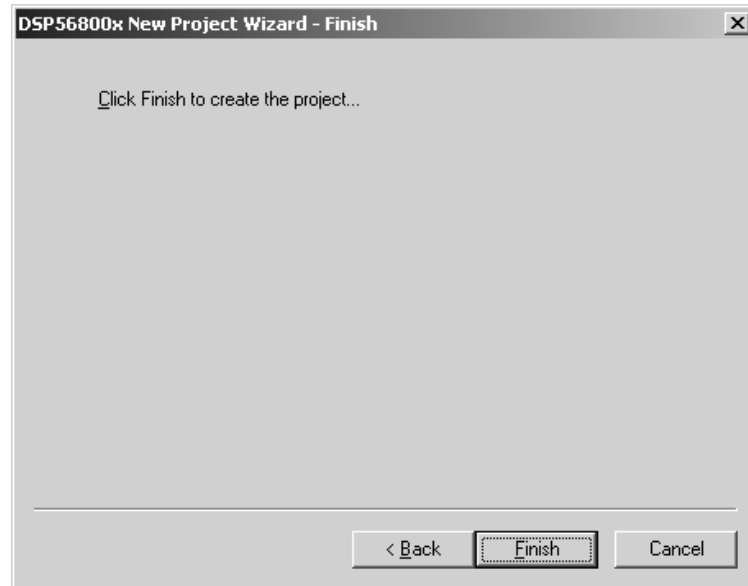
6. Select the target board and processor
  - a. Select the family, such as Simulators, from the **DSP56800x Family** list.
  - b. Select a processor or simulator, such as DSP56800E\_simulator, from the **Processors** list.
7. Click **Next**. The **DSP56800x New Project Wizard — Program Choice** dialog box (Figure 2.5) appears.

**Figure 2.5 DSP56800x New Project Wizard — Program Choice Dialog Box**



8. Select the example main[] program for this project, such as Simple C.
9. Click **Next**. The **DSP56800x New Project Wizard — Finish** dialog box (Figure 2.6) appears.

**Figure 2.6 DSP56800x New Project Wizard — Finish Dialog Box**



10. Click **Finish** to create the new project.

---

**NOTE** For more details of the DSP56800x new project wizard, please see Appendix B.

---

This completes project creation. You are ready to edit project contents, according to the optional steps below.

---

**NOTE** Stationery projects include source files that are placeholders for your own files. If a placeholder file has the same name as your file (such as `main.c`), you must replace the placeholder file with your source file.

---

11. (Optional) Remove files from the project.
- In the project window, select (highlight) the files.
  - Press the **Delete** key (or right-click the filename, then select Remove from the context menu). The filenames disappear.



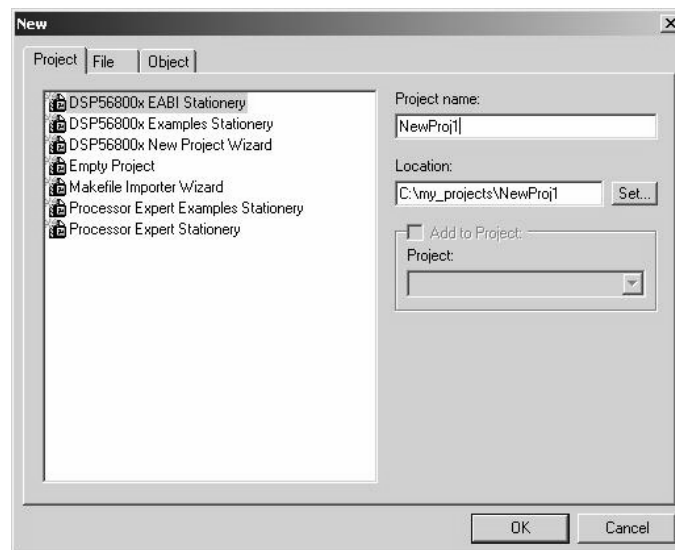
12. (Optional) Add source files to the project.
  - a. Method 1: From the main-window menu bar, select **Project>Add Files**. Then use the **Select files to add** dialog box to specify the files.
  - b. Method 2: Drag files from the desktop or Windows Explorer to the project window.
13. (Optional) Edit code in the source files.
  - a. Double-click the filename in the project window (or select the filename, then press the Enter key).
  - b. The IDE opens the file in the editor window; you are ready to edit file contents.

## Creating a New Project with the DSP56800x EABI Stationery

To create a sample project. Follow these steps:

1. From the menu bar, select **File>New**. The **New** window (Figure 2.7) appears.

**Figure 2.7 New Window**



## Getting Started Creating a Project

---

2. Specify a new DSP56800E project named NewProj1.
  - a. If necessary, click the Project tab to move the Project page to the front of the window.
  - b. From the project list, select (highlight) DSP56800E EABI Stationery.

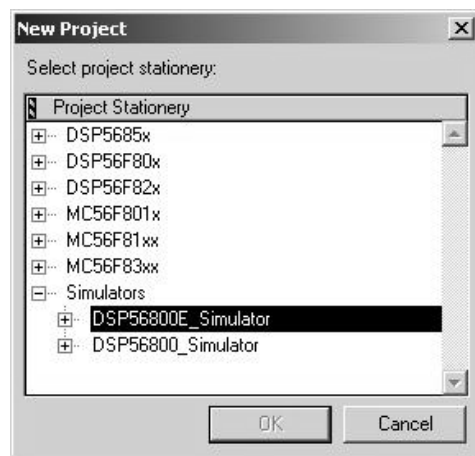
---

**NOTE** Stationery is a set of project templates, including libraries and placeholders for source code. Using stationery is the quickest way to create a new project.

---

- c. In the Project name text box, type: NewProj1. (When you save this project, the IDE automatically will add the .mcp extension to its filename.)
3. In the **New** window, click the OK button. The **New Project** window (Figure 2.8) appears, listing board-specific project stationery.

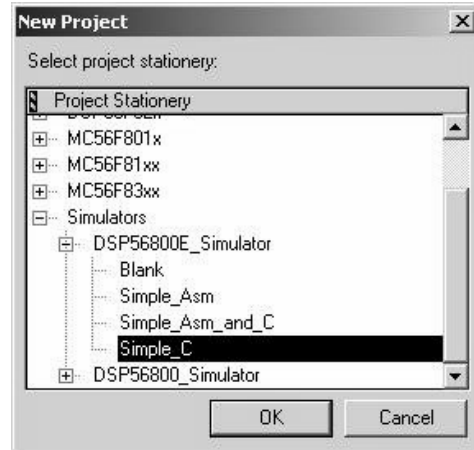
**Figure 2.8 New Project Window**



4. Select the simulator C stationery target.
  - a. Click the expand control (+) for the M56800E Simulator. The tree expands to show stationery selections.

- b. Select (highlight) Simple C. (Figure 2.9 shows this selection.)

**Figure 2.9 Simulator Simple C Selection**



---

**NOTE** You should select a simulator target if your system is not connected to a development board. If you do have a development board, your target selection must correspond to the board's processor.

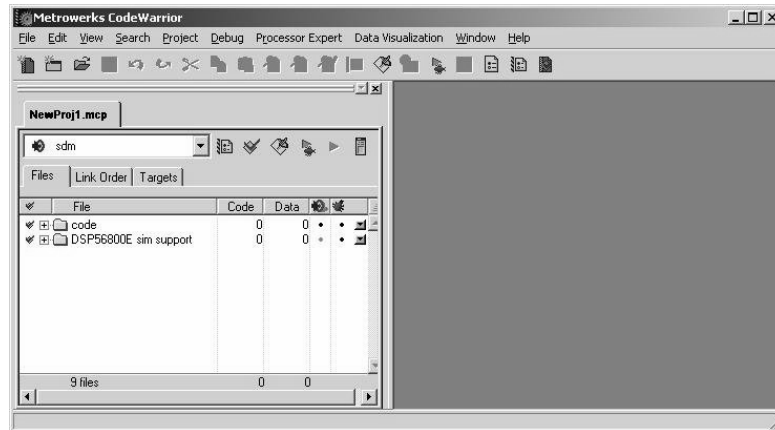
---

- c. Click the **OK** button. A project window opens, listing the folders for project NewProj1.mcp. Figure 2.10 shows this project window docked in the IDE main window.

## Getting Started Creating a Project

---

**Figure 2.10 Project Window (docked)**



---

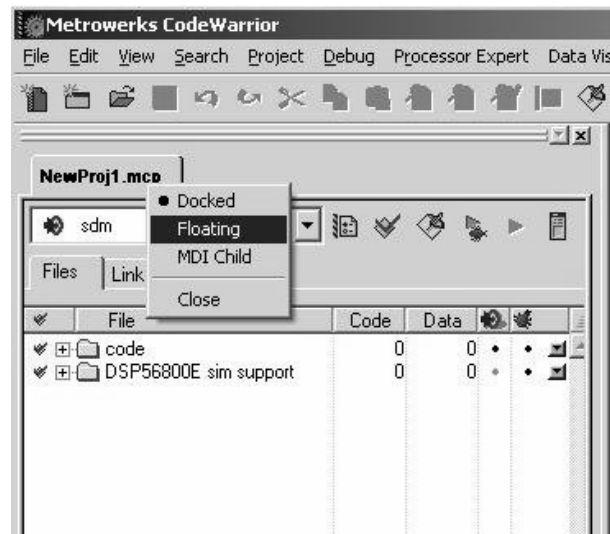
**NOTE** The IDE has the same functionality whether subordinate windows (such as the project window) are docked, floating, or child.

To undock the project window, right-click its title tab, then select Floating or Child from the context menu. Figure 2.11 shows this selection.

To dock a floating window, right-click its title bar, then select Docked from the context menu.

---

**Figure 2.11 Project Window (docked)**



5. This completes project creation. You are ready to edit project contents, according to the optional steps below.

<b>NOTE</b>	Stationery projects include source files that are placeholders for your own files. If a placeholder file has the same name as your file (such as <code>main.c</code> ), you must remove the placeholder file before adding your source file.
-------------	--

6. (Optional) Remove files from the project.
  - a. In the project window, select (highlight) the files.
  - b. Press the **Delete** key (or right-click the filename, then select Remove from the context menu). The filenames disappear.
7. (Optional) Add source files to the project.
  - a. Method 1: From the main-window menu bar, select **Project>Add Files**. Then use the **Select files to add** dialog box to specify the files.

# Freescale Semiconductor, Inc.

## Getting Started *Creating a Project*

---

- b. Method 2: Drag files from the desktop or Windows Explorer to the project window.
- 8. (Optional) Edit code in the source files.
  - a. Double-click the filename in the project window (or select the filename, then press the Enter key).
  - b. The IDE opens the file in the editor window; you are ready to edit file contents.

# Development Studio Overview

---

This chapter describes the CodeWarrior™ IDE and explains application development using the IDE. This chapter contains these sections:

- CodeWarrior IDE
- Development Process

If you are an experienced CodeWarrior IDE user, you will recognize the look and feel of the user interface. However, you must become familiar with the DSP56800E runtime software environment.

## CodeWarrior IDE

The CodeWarrior IDE lets you create software applications. It controls the project manager, the source-code editor, the class browser, the compiler, linker, and the debugger.

You use the project manager to organize all the files and settings related to your project. You can see your project at a glance and easily navigate among source-code files. The CodeWarrior IDE automatically manages build dependencies.

A project can have multiple build targets. A build target is a separate build (with its own settings) that uses some or all of the files in the project. For example, you can have both a debug version and a release version of your software as separate build targets within the same project.

The CodeWarrior IDE has an extensible architecture that uses plug-in compilers and linkers to target various operating systems and microprocessors. The CodeWarrior CD includes a C compiler for the DSP56800E family of processors. Other CodeWarrior software packages include C, C++, and Java compilers for Win32, Mac® OS, Linux, and other hardware and software combinations.

The IDE includes:

## Development Studio Overview

### Development Process

---

- **CodeWarrior Compiler for DSP56800E** — an ANSI-compliant C compiler, based on the same compiler architecture used in all CodeWarrior C compilers. Use this compiler with the CodeWarrior linker for DSP56800E to generate DSP56800E applications and libraries.

---

<b>NOTE</b>	The CodeWarrior compiler for DSP56800E does not support C++.
-------------	--

---

- **CodeWarrior Assembler for DSP56800E** — an assembler that features easy-to-use syntax. It assembles any project file that has a .asm filename extension. For further information, refer to the *Code Warrior Development Studio Freescale DSP56800x Embedded Systems Assembler Manual*.
- **CodeWarrior Linker for DSP56800E** — a linker that lets you generate either Executable and Linker Format (ELF) or S-record output files for your application.
- **CodeWarrior Debugger for DSP56800E** — a debugger that controls your program's execution, letting you see what happens internally as your program runs. Use this debugger to find problems in your program.

The debugger can execute your program one statement at a time, suspending execution when control reaches a specified point. When the debugger stops a program, you can view the chain of function calls, examine and change the values of variables, inspect processor register contents, and see the contents of memory.

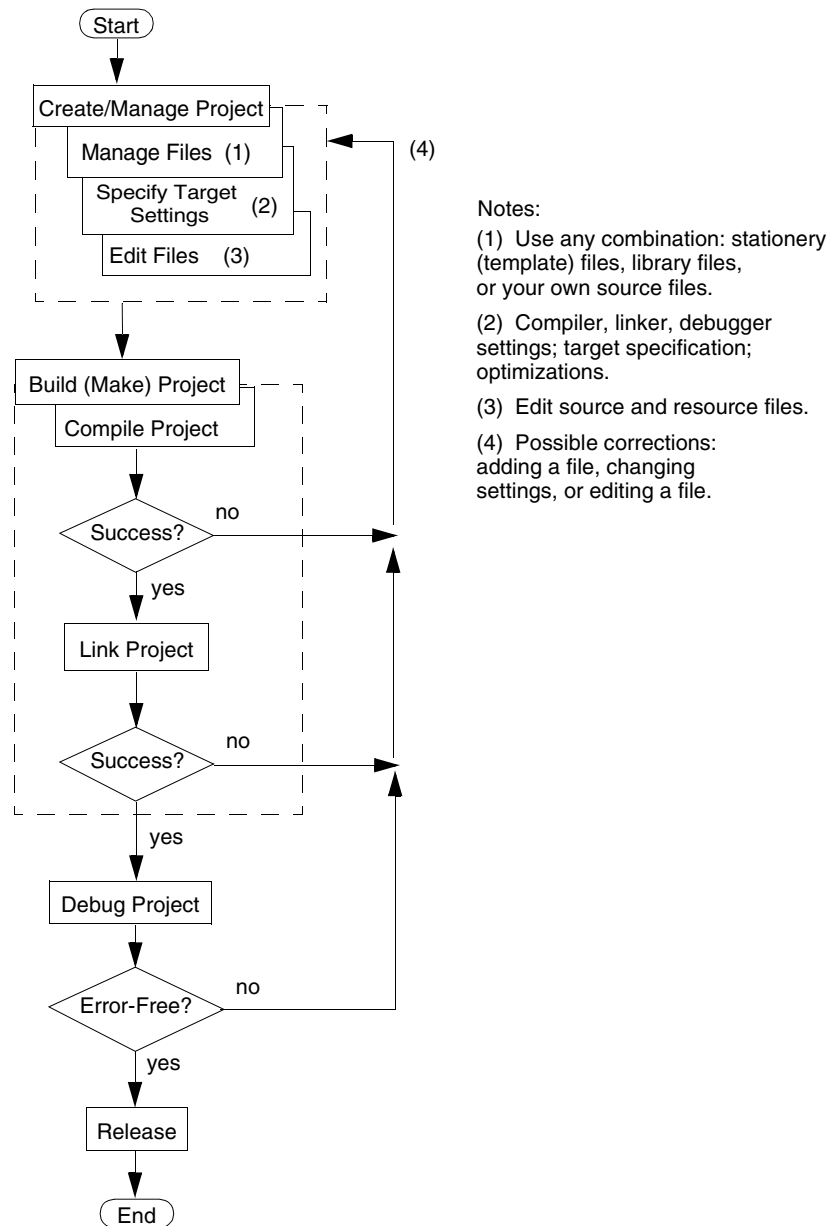
- **Metrowerks Standard Library (MSL)** — a set of ANSI-compliant, standard C libraries for use in developing DSP56800E applications. Access the library sources for use in your projects. A subset of those used for all platform targets, these libraries are customized and the runtime adapted for DSP56800E development.

## Development Process

The CodeWarrior IDE helps you manage your development work more effectively than you can with a traditional command-line environment. Figure 3.1 depicts application development using the IDE.



### Figure 3.1 CodeWarrior IDE Application Development

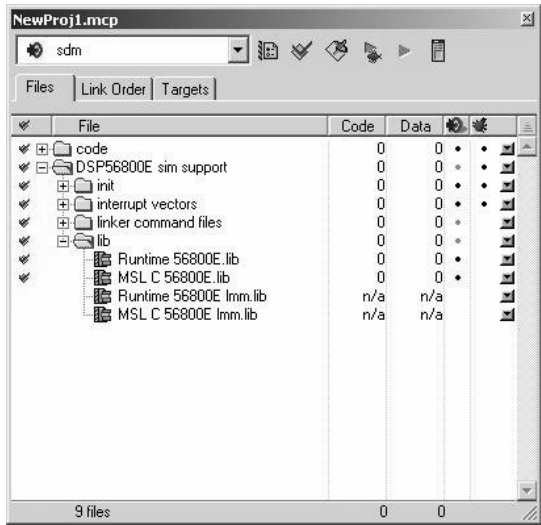


## Project Files

A CodeWarrior project consists of source-code, library, and other files. The project window (Figure 3.2) lists all files of a project, letting you:

- Add files,
- Remove files,
- Specify the link order,
- Assign files to build targets, and
- Direct the IDE to generate debug information for files.

Figure 3.2 Project Window



**NOTE** Figure 3.2 shows a floating project window. Alternatively, you can dock the project window in the IDE main window or make it a child window. You can have multiple project windows open at the same time; if the windows are docked, their tabs let you control which one is at the front of the main window.

The CodeWarrior IDE automatically handles the dependencies among project files, and stores compiler and linker settings for each build target. The IDE tracks which files have changed since your last build, recompiling only those files during your next project build.

A CodeWarrior project is analogous to a collection of makefiles, as the same project can contain multiple builds. Examples are a debug version and a release version of code, both part of the same project. As earlier text explained, *build targets* are such different builds within a single project.

## Editing Code

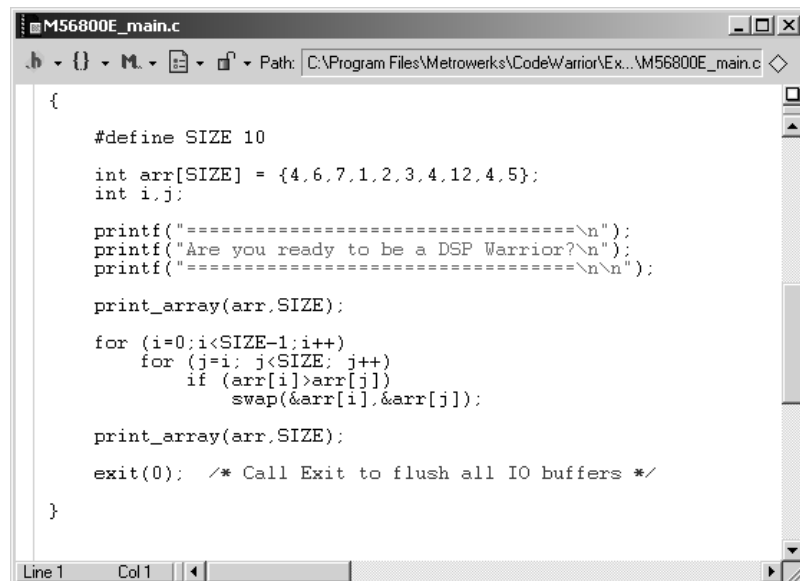
The CodeWarrior text editor handles text files in MS-DOS®, Windows®, UNIX, and Mac® OS formats.

To edit a source-code file (or any other editable project file), either:

- Double-click its filename in the project window, or
- Select (highlight) the filename, then drag the highlighted filename to the CodeWarrior main window.

The IDE opens the file in the editor window (Figure 3.3). This window lets you switch between related files, locate particular functions, mark locations within a file, or go to a specific line of code.

**Figure 3.3 Editor Window**



---

**NOTE** Figure 3.3 shows a floating editor window. Alternatively, you can dock the editor window in the IDE main window or make it a child window.

---

## Building: Compiling and Linking

For the CodeWarrior IDE, *building* includes both compiling and linking. To start building, you select **Project>Make**, from the IDE main-window menu bar. The IDE compiler:

- Generates an object-code file from each source-code file of the build target, incorporating appropriate optimizations.
- Updates other files of the build target, as appropriate.
- In case of errors, issues appropriate error messages and halts.

---

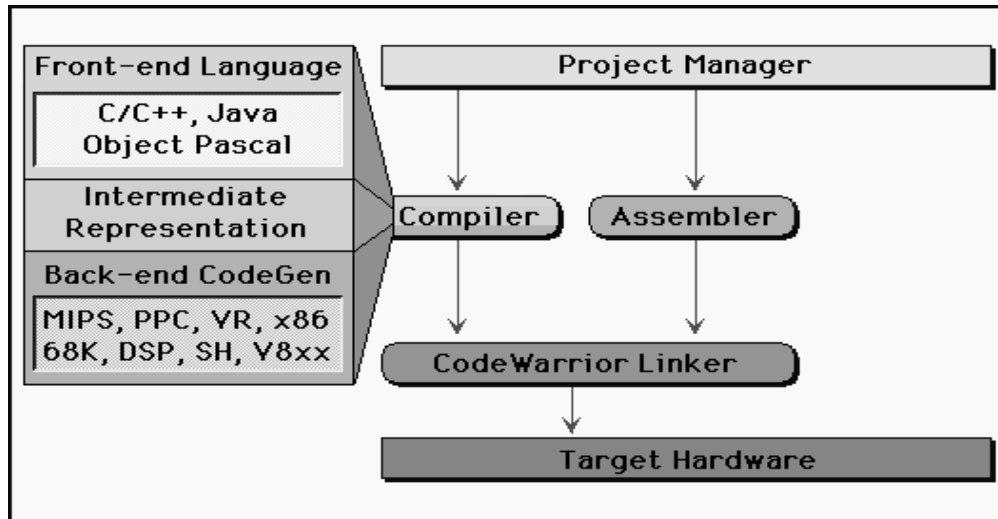
**NOTE** It is possible to compile a single source file. To do so, highlight its filename in the project window, then select **Project > Compile**, from the main-window menu bar. Another useful option is compiling all modified files of the build target: select **Project>Bring Up to Date** from the main-window menu bar.

---

In UNIX and other command-line environments, the IDE stores object code in a binary ( `.o` or `.obj` ) file. On Windows targets, the IDE stores and manages object files internally in the data folder.

A proprietary compiler architecture at the heart of the CodeWarrior IDE handles multiple languages and platform targets. Front-end language compilers generate an intermediate representation (IR) of syntactically correct source code. This IR is memory-resident and language-independent. Back-end compilers generate code from the IR for specific platform targets. As Figure 3.4 depicts, the CodeWarrior IDE manages this whole process.

**Figure 3.4 CodeWarrior Build System**



This architecture means that the CodeWarrior IDE uses the same front-end compiler to support multiple back-end platform targets. In some cases, the same back-end compiler can generate code from a variety of languages. User benefits of this architecture include:

- An advance in the C/C++ front-end compiler means an immediate advance in all code generation.
- Optimizations in the IR mean that any new code generator is highly optimized.
- Targeting a new processor does not require compiler-related changes in source code, simplifying porting.

Metrowerks builds all compilers as plug-in modules. The compiler and linker components are modular plug-ins. Metrowerks publishes this API, so that developers can create custom or proprietary tools. For more information, go to Metrowerks Support:

<http://www.metrowerks.com/MW/Support>

When compilation succeeds, building moves on to linking. The IDE linker:

- Links the object files into one executable file. (You use the M56800E Target settings panel to name the executable file.)
- In case of errors, issues appropriate error messages and halts.

The IDE uses linker command files to control the linker, so you do not need to specify a list of object files. The Project Manager tracks all the object files automatically; it lets you specify the link order.

When linking succeeds, you are ready to test and debug your application.

## Debugging

To debug your application, select **Project>Debug** from the main-window menu bar. The debugger window opens, displaying your program code.

Run the application from within the debugger, to observe results. The debugger lets you set breakpoints, and check register, parameter, and other values at specific points of code execution.

When your code executes correctly, you are ready to add features, to release the application to testers, or to release the application to customers.

---

<b>NOTE</b>	Another debugging feature of the CodeWarrior IDE is viewing preprocessor output. This helps you track down bugs caused by macro expansion or another subtlety of the preprocessor. To use this feature, specify the output filename in the project window, then select <b>Project&gt;Preprocess</b> from the main-window menu bar. A new window opens to show the preprocessed file.
-------------	--

---

# Target Settings

---

Each build target in a CodeWarrior™ project has its own settings. This chapter explains the target settings panels for DSP56800E software development. The settings that you select affect the DSP56800E compiler, linker, assembler, and debugger.

This chapter contains the following sections:

- Target Settings Overview
- CodeWarrior IDE Target Settings Panels
- DSP56800E-Specific Target Settings Panels

## Target Settings Overview

The target settings control:

- Compiler options
- Linker options
- Assembler options
- Debugger options
- Error and warning messages

When you create a project using stationery, the build targets, which are part of the stationery, already include default target settings. You can use those default target settings (if the settings are appropriate), or you can change them.

---

<b>NOTE</b>	Use the DSP56800E project stationery when you create a new project.
-------------	---

---

## Target Setting Panels

Table 4.1 lists the target settings panels:

# Freescale Semiconductor, Inc.

## Target Settings

### Target Settings Overview

- Links identify the panels specific to DSP56800E projects. Click the link to go to the explanation of that panel.
- The Use column explains the purpose of generic IDE panels that also can apply to DSP56800E projects. For explanations of these panels, see the *IDE User Guide*.

**Table 4.1 Target Setting Panels**

Group	Panel Name	Use
Target	Target Settings	
	Access Paths	Selects the paths that the IDE searches to find files of your project. Types include absolute and project-relative.
	Build Extras	Sets options for building a project, including using a third-party debugger.
	File Mappings	Associates a filename extension, such as .c, with a plug-in compiler.
	Source Trees	Defines project -specific source trees (root paths) for your project.
	M56800E Target	
Language Settings	C/C++ Language (C Only)	
	C/C++ Preprocessor	
	C/C++ Warnings	
	M56800E Assembler	
Code Generation	ELF Disassembler	
	M56800E Processor	
	Global Optimization	Configures how the compiler optimizes code.
Linker	M56800E Linker	
Editor	Custom Keywords	Changes colors for different types of text.



**Table 4.1 Target Setting Panels (*continued*)**

Group	Panel Name	Use
Debugger	Debugger Settings	Specifies settings for the CodeWarrior debugger.
	Remote Debugging	
	M56800E Target (Debugging)	
	Remote Debug Options	

## Changing Target Settings

To change target settings:

1. Select **Edit > Target Name Settings**.

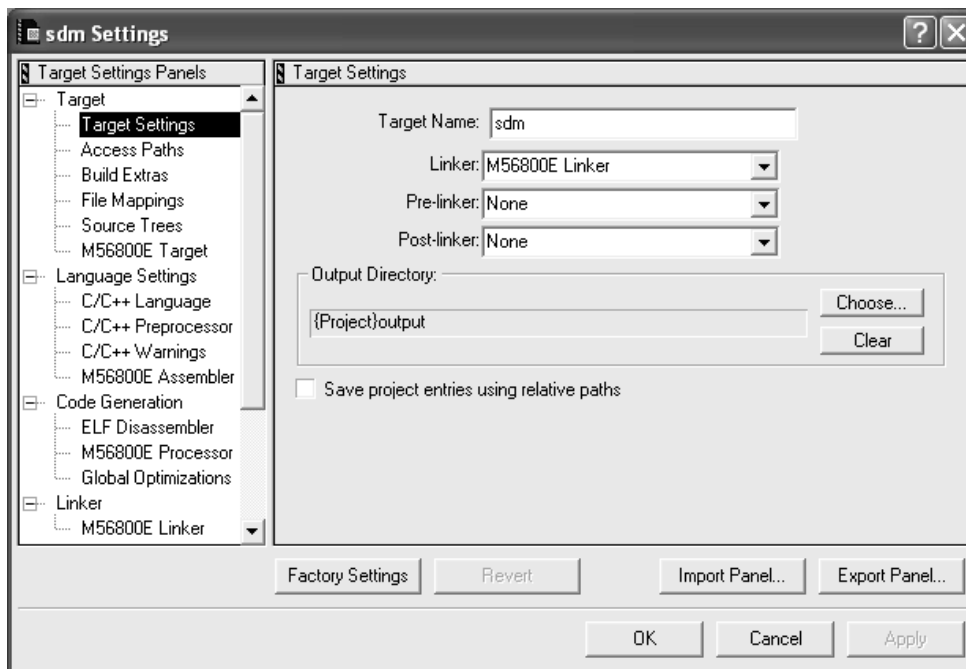
*Target* is the name of the current build target in the CodeWarrior project.

After you select this menu item, the CodeWarrior IDE displays the **Target Settings** window (Figure 4.1).

## Target Settings *Target Settings Overview*

---

**Figure 4.1 Target Settings Window**



The left side of the **Target Settings** window contains a list of target settings panels that apply to the current build target.

2. To view the Target Settings panel:

Click on the name of the **Target Settings** panel in the **Target Settings** panels list on the left side of the **Target Settings** window.

The CodeWarrior IDE displays the target settings panel that you selected.

3. Change the settings in the panel.
4. Click OK.

## Exporting and Importing Panel Options to XML Files

The CodeWarrior IDE can export options for the current settings panel to an Extensible Markup Language (XML) file or import options for the current settings panel from a previously saved XML file.

### Exporting Panel Options to XML File

1. Click the Export Panel button.
2. Assign a name to the XML file and save the file in the desired location.

### Importing Panel Options from XML File

1. Click the Import Panel button.
2. Locate the XML file to where you saved the options for the current settings panel.
3. Open the file to import the options.

### Saving New Target Settings in Stationery

To create stationery files with new target settings:

1. Create your new project from an existing stationery.
2. Change the target settings in your new project for any or all of the build targets in the project.
3. Save the new project in the Stationery folder.

## Restoring Target Settings

After you change settings in an existing project, you can restore the previous settings by using any of the following methods:

- To restore the previous settings, click **Revert** at the bottom of the **Target Settings** window.
- To restore the settings to the factory defaults, click **Factory Settings** at the bottom of the window.

## CodeWarrior IDE Target Settings Panels

Table 4.2 lists and explains the CodeWarrior IDE target settings panels that can apply to DSP56800E.

**Table 4.2 Code Warrior IDE Target Settings Panels**

Target Settings Panels	Description
Access Paths	Use this panel to select the paths that the CodeWarrior IDE searches to find files in your project. You can add several kinds of paths including absolute and project-relative. <i>See IDE User Guide.</i>
Build Extras	Use this panel to set options that affect the way the CodeWarrior IDE builds a project, including the use of a third-party debugger. <i>See IDE User Guide.</i>
File Mappings	Use this panel to associate a file name extension, such as .c, with a plug-in compiler. <i>See IDE User Guide.</i>
Source Trees	Use this panel to define project-specific source trees (root paths) for use in your projects. <i>See IDE User Guide.</i>
Custom Keywords	Use this panel to change the colors that the CodeWarrior IDE uses for different types of text. <i>See IDE User Guide.</i>
Global Optimizations	Use this panel to configure how the compiler optimizes the object code. <i>See IDE User Guide.</i>
Debugger Settings	Use this panel to specify settings for the CodeWarrior debugger.

## DSP56800E-Specific Target Settings Panels

The rest of this chapter explains the target settings panels specific to DSP56800E development.

## Target Settings

Use the **Target Settings** panel (Figure 4.2) to specify a linker. This selection also specifies your target. Table 4.3 explains the elements of the Target Settings panel.

The Target Settings window changes its list of panels to reflect your linker choice. As your linker choice determines which other panels are appropriate, it should be your first settings action.

Figure 4.2 Target Settings Panel

Table 4.3 Target Settings Panel Elements

Element	Purpose	Comments
Target Name text box	Sets or changes the name of a build target.	For your development convenience, not the name of the final output file. (Use the AGB Target Setting panel to name the output file.)
Linker list box	Specifies the linker.	Select M56800E Linker.
Pre-linker list box	Specifies a pre-linker.	Select None. (No pre-linker is available for the M56800E linker.)
Post-linker list box	Specifies a post-linker.	Select None. (No post-linker is available for the M56800E linker.)

## Target Settings

### DSP56800E-Specific Target Settings Panels

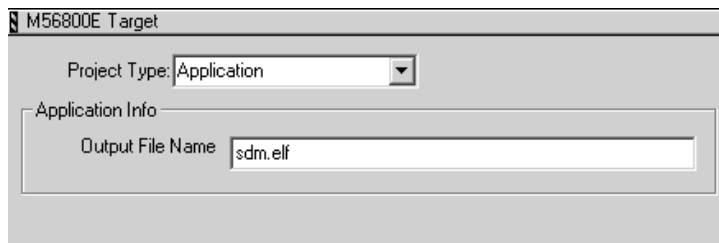
**Table 4.3 Target Settings Panel Elements (continued)**

Element	Purpose	Comments
Output Directory text box	Tells the IDE where to save the executable file. To specify a different output directory, click the <b>Choose</b> button, then use the access-path dialog box to specify a directory. (To delete such an alternate directory, click the <b>Clear</b> button.)	Default: the directory that contains the project file.
Save Project Entries Using Relative Paths checkbox	Controls whether multiple project files can have the same name: <ul style="list-style-type: none"> <li>Clear — Each project entry must have a unique name.</li> <li>Checked — The IDE uses relative paths to save project entries; entry names need not be unique.</li> </ul>	Default: Clear — project entries must have unique names.

## M56800E Target

Use the **M56800E Target** panel (Figure 4.3) to specify the project type and the name of the output file. Table 4.4 explains the elements of this panel.

**Figure 4.3 M56800E Target Panel**



**Table 4.4 M56800E Target Panel Elements**

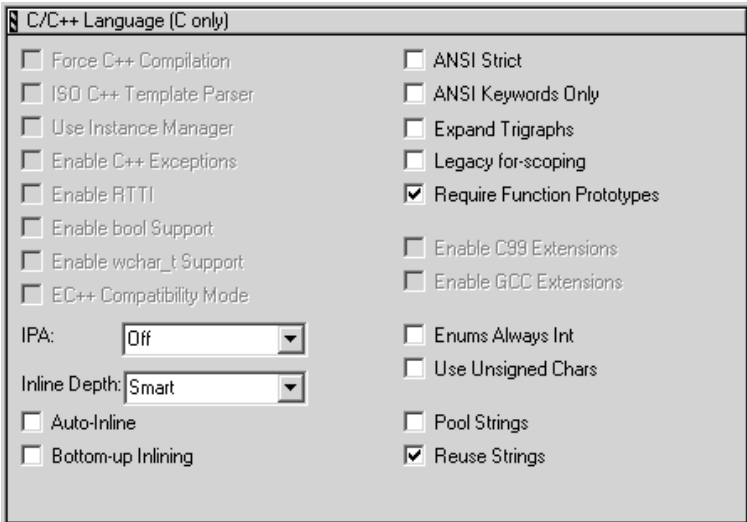
Element	Purpose	Comments
Project Type list box	Specifies an <b>Application</b> or <b>Library</b> project.	Application is the usual selection.
Output File Name text box	Specifies the name of the output file.	End application filenames with the .elf extension; end library filenames with the .lib extension.

**NOTE** Be sure to name libraries with the extension `.lib`. It is possible to use a different extension, but this requires a file-mapping entry in the **File Mappings** panel. For more information, see the *IDE User Guide*.

### C/C++ Language (C Only)

Use the **C/C++ Language (C Only)** panel (Figure 4.4) to specify C language features. Table 4.5 explains the elements of this panel that apply to the DSP56800E processor, which supports only the C language.

Figure 4.4 C/C++ Language Panel (C Only)



**NOTE** Always disable the following options, which do not apply to the DSP56800E compiler: Legacy for-scoping and Pool Strings.

# Freescale Semiconductor, Inc.

## Target Settings

DSP56800E-Specific Target Settings Panels

**Table 4.5 C/C++ Language (C Only) Panel Elements**

Element	Purpose	Comments
IPA list box	Specifies Interprocedural Analysis (IPA): Off — IPA is disabled File — inlining is deferred to the end of the file processing	
Inline Depth list box	Together with the ANSI Keyword Only checkbox, specifies whether to inline functions: Don't Inline — do not inline any Smart — inline small functions to a depth of 2 to 4 1 to 8 — Always inline functions to the number's depth Always inline — inline all functions, regardless of depth	If you call an inline function, the compiler inserts the function code, instead of issuing calling instructions. Inline functions execute faster, as there is no call. But overall code may be larger if function code is repeated in several places.
Auto-Inline checkbox	Checked — Compiler selects the functions to inline Clear — Compiler does not select functions for inlining	To check whether automatic inlining is in effect, use the <code>__option(auto_inline)</code> command.
Bottom-up Inlining checkbox	Checked — For a chain of function calls, the compiler begins inlining with the last function. Clear — Compiler does not do bottom-up inlining.	To check whether bottom-up inlining is in effect, use the <code>__option(inline_bottom_up)</code> command.
ANSI Strict checkbox	Checked — Disables CodeWarrior compiler extensions to C Clear — Permits CodeWarrior compiler extensions to C	Extensions are C++-style comments, unnamed arguments in function definitions, # not and argument in macros, identifier after #endif, typecasted pointers as lvalues, converting pointers to same-size types, arrays of zero length in structures, and the D constant suffix. To check whether ANSI strictness is in effect, use the <code>__option(ANSI_strict)</code> command.



**Table 4.5 C/C++ Language (C Only) Panel Elements (*continued*)**

Element	Purpose	Comments
ANSI Keywords Only checkbox	Checked — Does not permit additional keywords of CodeWarrior C. Clear — Does permit additional keywords.	Additional keywords are asm (use the compiler built-in assembler) and inline (lets you declare a C function to be inline). To check whether this keyword restriction is in effect, use the <code>__option(only_std_keywords)</code> command.
Expand Trigraphs checkbox	Checked — C Compiler ignores trigraph characters. Clear — C Compiler does not allow trigraph characters, per strict ANSI/ISO standards.	Many common character constants resemble trigraph sequences, especially on the Mac OS. This extension lets you use these constants without including escape characters. NOTE: If this option is on, be careful about initializing strings or multi-character constants that include question marks. To check whether this option is on, use the <code>__option(trigraphs)</code> command.
Require Function Prototypes checkbox	Checked — Compiler does not allow functions that do not have prototypes. Clear — Compiler allows functions without prototypes.	This option helps prevent errors from calling a function before its declaration or definition. To check whether this option is in effect, use the <code>__option(require_prototypes)</code> command.
Enums Always Int checkbox	Checked — Restricts all enumerators to the size of a signed int. Clear — Compiler converts unsigned int enumerators to signed int, then chooses an accommodating data type, char to long int.	To check whether this restriction is in effect, use the <code>__option(enumalwaysint)</code> command.

## Target Settings

*DSP56800E-Specific Target Settings Panels*

**Table 4.5 C/C++ Language (C Only) Panel Elements (*continued*)**

Element	Purpose	Comments
Use Unsigned Chars checkbox	Checked — Compiler treats a char declaration as an unsigned char declaration. Clear — Compiler treats char and unsigned char declarations differently.	Some libraries were compiled without this option. Selecting this option may make your code incompatible with such libraries. To check whether this option is in effect, use the <code>__option(unsigned_char)</code> command.
Reuse Strings checkbox	Checked — Compiler stores only one copy of identical string literals, saving memory space. Clear — Compiler stores each string literal.	If you select this option, changing one of the strings affects them all.

## C/C++ Preprocessor

The C/C++ Preprocessor (Figure 4.5) panel controls how the preprocessor interprets source code. By modifying the settings on this panel, you can control how the preprocessor translates source code into preprocessed code.

More specifically, the C/C++ Preprocessor panel provides an editable text field that can be used to `#define` macros, set `#pragmas`, or `#include` prefix files.

**Figure 4.5 The C/C++ Preprocessor Panel**

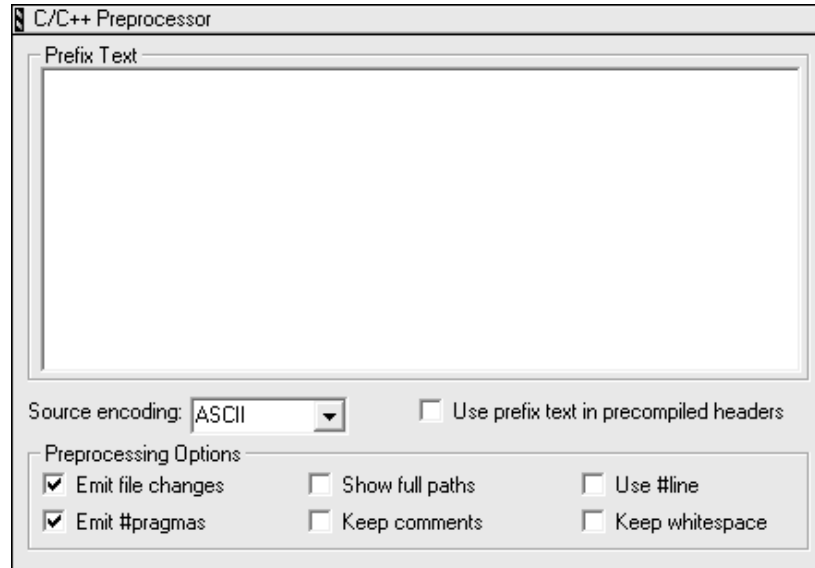


Table 4.6 provides information about the options in this panel.

**Table 4.6 C/C++ Language Preprocessor Elements**

Element	Purpose	Comments
Source encoding	Allows you to specify the default encoding of source files. Multibyte and Unicode source text is supported.	To replicate the obsolete option "Multi-Byte Aware", set this option to System or Autodetect. Additionally, options that affect the "preprocess" request appear in this panel.
Use prefix text in precompiled header	Controls whether a *.pch or *.pch++ file incorporates the prefix text into itself.	This option defaults to "off" to correspond with previous versions of the compiler that ignore the prefix file when building precompiled headers. If any #pragmas are imported from old C/C++ Language (C Only) Panel settings, this option is set to "on".
Emit file changes	Controls whether notification of file changes (or #line changes) appear in the output.	

## Target Settings

DSP56800E-Specific Target Settings Panels

**Table 4.6 C/C++ Language Preprocessor Elements (*continued*)**

Element	Purpose	Comments
Emit #pragmas	Controls whether #pragmas encountered in the source text appear in the preprocessor output.	This option is essential for producing reproducible test cases for bug reports.
Show full paths	Controls whether file changes show the full path or the base filename of the file.	
Keep comments	Controls whether comments are emitted in the output.	
Use #line	Controls whether file changes appear in comments (as before) or in #line directives.	
Keep whitespace	Controls whether whitespace is stripped out or copied into the output.	This is useful for keeping the starting column aligned with the original source, though we attempt to preserve space within the line. This doesn't apply when macros are expanded.

## C/C++ Warnings

Use the C/C++ Warnings panel (Figure 4.6) to specify C language features for the DSP56800E. Table 4.7 explains the elements of this panel.

---

**NOTE** The CodeWarrior compiler for DSP56800E does not support C++.

---

Figure 4.6 C/C++ Warnings Panel

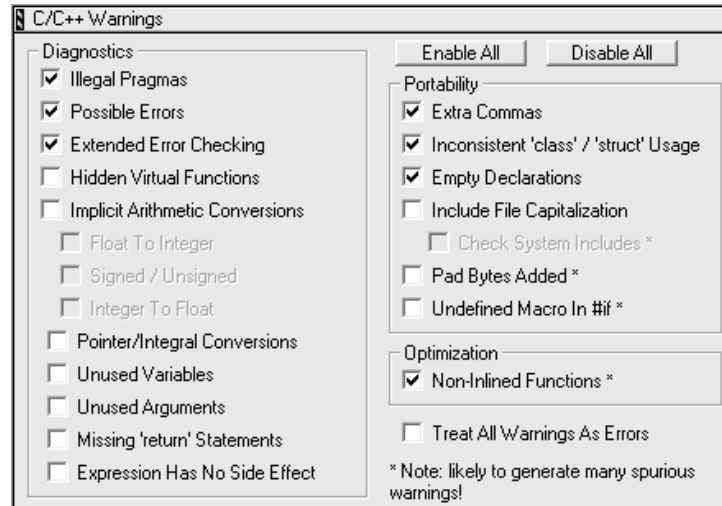


Table 4.7 C/C++ Warnings Panel Elements

Element	Purpose	Comments
Illegal Pragmas checkbox	Checked — Compiler issues warnings about invalid pragma statements. Clear — Compiler does not issue such warnings.	According to this option, the invalid statement <b>#pragma near_data off</b> would prompt the compiler response <b>WARNING: near data is not a pragma</b> . To check whether this option is in effect, use the <code>__option(warn_illpragma)</code> command.
Possible Errors checkbox	Checked — Compiler checks for common typing mistakes, such as <code>==</code> for <code>=</code> . Clear — Compiler does not perform such checks.	If this option is in effect, any of these conditions triggers a warning: an assignment in a logical expression; an assignment in a while, if, or for expression; an equal comparison in a statement that contains a single expression; a semicolon immediately after a while, if, or for statement. To check whether this option is in effect, use the <code>__option(warn_possunwant)</code> command.

## Target Settings

DSP56800E-Specific Target Settings Panels

**Table 4.7 C/C++ Warnings Panel Elements (continued)**

Element	Purpose	Comments
Extended Error Checking checkbox	Checked — Compiler issues warnings in response to specific syntax problems. Clear — Compiler does not perform such checks.	Syntax problems are: a non-void function without a return statement, an integer or floating-point value assigned to an enum type, or an empty return statement in a function not declared void. To check whether this option is in effect, use the <code>__option(extended_errorcheck)</code> command.
Hidden Virtual Functions	Leave clear.	Does not apply to C.
Implicit Arithmetic Conversions checkbox	Checked — Compiler verifies that operation destinations are large enough to hold all possible results. Clear — Compiler does not perform such checks.	If this option is in effect, the compiler would issue a warning in response to assigning a long value to a char variable. To check whether this option is in effect, use the <code>__option(warn_implicitconv)</code> command.
Pointer/Integral Conversions	Checked — Compiler checks for pointer/integral conversions. Clear — Compiler does not perform such checks.	See <code>#pragma warn_any_ptr_int_conv</code> and <code>#pragma warn_ptr_int_conv</code> .
Unused Variables checkbox	Checked — Compiler checks for declared, but unused, variables. Clear — Compiler does not perform such checks.	The pragma <b>unused</b> overrides this option. To check whether this option is in effect, use the <code>__option(warn_unusedvar)</code> command.
Unused Arguments checkbox	Checked — Compiler checks for declared, but unused, arguments. Clear — Compiler does not perform such checks.	The pragma <b>unused</b> overrides this option. Another way to override this option is clearing the ANSI Strict checkbox of the C/C++ Language (C Only) panel, then not assigning a name to the unused argument. To check whether this option is in effect, use the <code>__option(warn_unusedarg)</code> command.

**Table 4.7 C/C++ Warnings Panel Elements (*continued*)**

Element	Purpose	Comments
Missing 'return' Statements	Checked — Compiler checks for missing 'return' statements. Clear — Compiler does not perform such checks.	See #pragma warn_missingreturn.
Expression Has No Side Effect	Checked — Compiler issues warning if expression has no side effect. Clear — Compiler does not perform such checks.	See #pragma warn_no_side_effect
Extra Commas checkbox	Checked — Compiler checks for extra commas in enums. Clear — Compiler does not perform such checks.	To check whether this option is in effect, use the <code>__option(warn_extracomma)</code> command.
Inconsistent Use of 'class' and 'struct' Keywords checkbox	Leave clear.	Does not apply to C.
Empty Declarations checkbox	Checked — Compiler issues warnings about declarations without variable names. Clear — Compiler does not issue such warnings.	According to this option, the incomplete declaration <code>int ;</code> would prompt the compiler response <b>WARNING</b> . To check whether this option is in effect, use the <code>__option(warn_emptydecl)</code> command.
Include File Capitalization	Checked — Compiler issues warning about include file capitalization. Clear — Compiler does not perform such checks.	See #pragma warn_filenameecaps.
Pad Bytes Added	Checked — Compiler checks for pad bytes added. Clear — Compiler does not perform such checks.	See #pragma warn_padding.
Undefined Macro In #if	Checked — Compiler checks for undefined macro in #if. Clear — Compiler does not perform such checks.	See #pragma warn_undefmacro.

## Target Settings

### DSP56800E-Specific Target Settings Panels

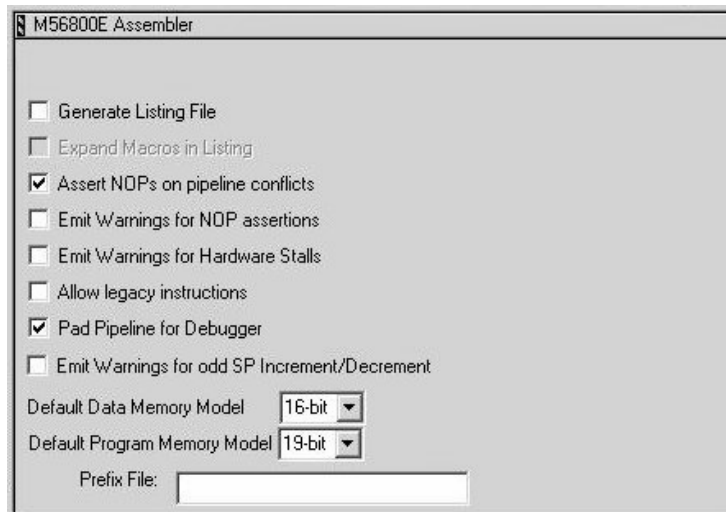
**Table 4.7 C/C++ Warnings Panel Elements (continued)**

Element	Purpose	Comments
Non-Inlined Functions checkbox	Checked — Compiler issues a warning if unable to inline a function. Clear — Compiler does not issue such warnings.	To check whether this option is in effect, use the <code>__option(warn_notinlined)</code> command.
Treat All Warnings As Errors checkbox	Checked — System displays warnings as error messages. Clear — System keeps warnings and error messages distinct.	

## M56800E Assembler

Use the M56800E Assembler panel (Figure 4.7) to specify the format of the assembly source files and the code that the DSP56800E assembler generates. Table 4.8 explains the elements of this panel.

**Figure 4.7 M56800E Assembler Panel**





**Table 4.8 M56800E Assembler Panel Elements**

Element	Purpose	Comments
Generate Listing File checkbox	Checked — Assembler generates a listing file during IDE assembly of source files. Clear — Assembler does not generate a listing file.	A listing file contains the source file with line numbers, relocation information, and macro expansions. The filename extension is .lst.
Expand Macros in Listing checkbox	Checked — Assembler macros expand in the assembler listing. Clear — Assembler macros do not expand.	This checkbox is available only if the Generate Listing File checkbox is checked.
Assert NOPs on pipeline conflicts checkbox	Checked — Assembler automatically resolves pipeline conflicts by inserting NOPs. Clear — Assembler does not insert NOPs; it reports pipeline conflicts in error messages.	
Emit Warnings for NOP Assertions checkbox	Checked — Assembler issues a warning any time it inserts a NOP to prevent a pipeline conflict. Clear — Assembler does not issue such warnings.	This checkbox is available only if the Assert NOPs on pipeline conflicts checkbox is checked.
Emit Warnings for Hardware Stalls checkbox	Checked — Assembler warns that a hardware stall will occur upon execution. Clear — Assembler does not issue such warnings.	This option helps optimize the cycle count.
Allow legacy instructions checkbox	Checked — Assembler permits legacy DSP56800 instruction syntax. Clear — Assembler does not permit this legacy syntax.	Selecting this option sets the Default Data Memory Model and Default Program Memory Model values to 16 bits.
Pad Pipeline for Debugger checkbox	Checked — Mandatory for using the debugger. Inserts NOPs after certain branch instructions to make breakpoints work reliably. Clear — Does not insert such NOPs.	If you select this option, you should select the same option in the M56800E Processor Settings panel. Selecting this option increases code size by 5 percent. But not selecting this option risks nonrecovery after the debugger comes to breakpoint branch instructions.
Emit Warnings for odd SP Increment/Decrement checkbox	Checked — Enables assembler warnings about instructions that could misalign the stack frame. Clear — Does not enable such warnings.	

## Target Settings

DSP56800E-Specific Target Settings Panels

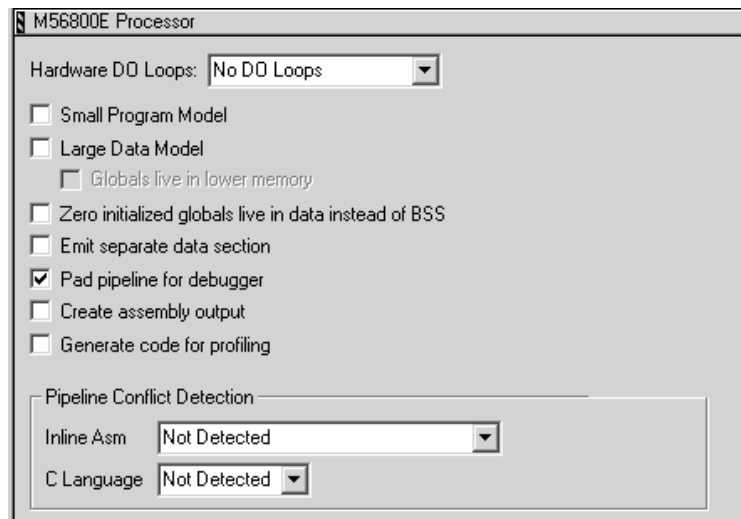
**Table 4.8 M56800E Assembler Panel Elements (continued)**

Element	Purpose	Comments
Default Data Memory Model list box	Specifies 16 or 24 bits as the default size.	Factory setting: 16 bits.
Default Program Memory Model list box	Specifies 16, 19, or 21 bits as the default size.	Factory setting: 19 bits.
Prefix File text box	Specifies a file to be included at the beginning of every assembly file of the project.	Lets you include common definitions without using an include directive in every file.

## M56800E Processor

Use the M56800E Processor panel (Figure 4.8) to specify the kind of code the compiler creates. This panel is available only if the current build target uses the M56800E Linker. Table 4.9 explains the elements of this panel.

**Figure 4.8 M56800E Processor Panel**



**Table 4.9 M56800E Processor Panel Elements**

Element	Purpose	Comments
Hardware DO Loops list box	Specifies the level of hardware DO loops: <ul style="list-style-type: none"> <li>No Do Loops — Compiler does not generate any</li> <li>No Nest DO Loops — Compiler generates hardware DO loops, but does not nest them</li> <li>Nested DO Loops — Compiler generates hardware Do loops, nesting them two deep.</li> </ul>	If hardware DO loops are enabled, debugging will be inconsistent about stepping into loops. Test immediately after this table contains additional Do-loop information.
Small Program Model checkbox	Checked — Compiler generates a more efficient switch table, provided that code fits into the range 0x0—0xFFFF Clear — Compiler generates an ordinary switch table.	Do not check this checkbox unless the entire program code fits into the 0x0—0xFFFF memory range.
Large Data Model checkbox	Checked — Extends DSP56800E addressing range by providing 24-bit address capability to instructions Clear — Does not extend address range	24-bit address modes allow access beyond the 64K-byte boundary of 16-bit addressing.
Globals live in lower memory checkbox	Checked — Compiler uses 24-bit addressing for pointer and stack operations, 16-bit addressing for access to global and static data. Clear — Compiler uses 24-bit addressing for all data access.	This checkbox is available only if the Large Data Model checkbox is checked.
Pad pipeline for debugger checkbox	Checked — Mandatory for using the debugger. Inserts NOPs after certain branch instructions to make breakpoints work reliably. Clear — Does not insert such NOPs.	If you select this option, you should select the same option in the M56800E Assembler panel. Selecting this option increases code size by 5 percent. But not selectins this option risks nonrecovery after the debugger comes to breakpoint branch instructions.
Emit separate character data section checkbox	Checked — Compiler breaks out all character data, placing it in appropriate data sections (.data.char, .bss.char, or .const.data.char). Clear — Compiler does not break out this data.	See additional information immediately after this table.

## Target Settings

DSP56800E-Specific Target Settings Panels

**Table 4.9 M56800E Processor Panel Elements (*continued*)**

Element	Purpose	Comments
Zero-initialized globals live in data instead of BSS checkbox	Checked — Globals initialized to zero reside in the .data section. Clear — Globals initialized to zero reside in the .bss section.	
Create Assembly Output checkbox	Checked — Assembler generates assembly code for each C file. Clear — Assembler does not generate assembly code for each C file.	The pragma #asmoutput overrides this option for individual files.
Generate code for profiling	Checked — Compiler generates code for profiling. Clear — Compiler does not generate code for profiling.	For more details about the profiler, see the “Profiler” on page 225.
Pipeline Conflict Detection Inline ASM list box	Specifies pipeline conflict detection during compiling of inline assembly source code: <ul style="list-style-type: none"> <li>• Not Detected — compiler does not check for conflicts</li> <li>• Conflict error — compiler issues error messages if it detects conflicts</li> <li>• Conflict Error/Hardware Stall Warning — compiler issues error messages if it detects conflicts, warnings if it detects hardware stalls</li> </ul>	For more information about pipeline conflicts, see the explanations of pragmas check_c_src_pipeline and check_inline_asm_pipeline.
Pipeline Conflict Detection C Language list box	Specifies pipeline conflict detection during compiling of C source code: <ul style="list-style-type: none"> <li>• Not Detected — compiler does not check for conflicts</li> <li>• Conflict error — compiler issues error messages if it detects conflicts</li> </ul>	For more information about pipeline conflicts, see the explanations of pragmas check_c_src_pipeline and check_inline_asm_pipeline.

The compiler generates hardware DO loops for two situations:

1. Aggregate (array and structure) initializations, and for struct copy, under any of these conditions:
  - The aggregate is byte aligned, and the aggregate size is greater than four bytes.
  - The aggregate is word aligned, and the aggregate size is greater than four words.

- The aggregate is long aligned, the aggregate size is greater than eight words, and the Global Optimizations panel specifies Optimize for Smaller Code Size.
  - The aggregate is long aligned, the aggregate size is greater than 32 words, and the Global Optimizations panel specifies Optimize for Faster Execution.
2. Counted loops in C, provided that the loop counter value is less than 65536, and that there are no jumps to subroutines inside the loop.

If you enable separate character data sections, the compiler puts character data (and structures containing character data) into these sections:

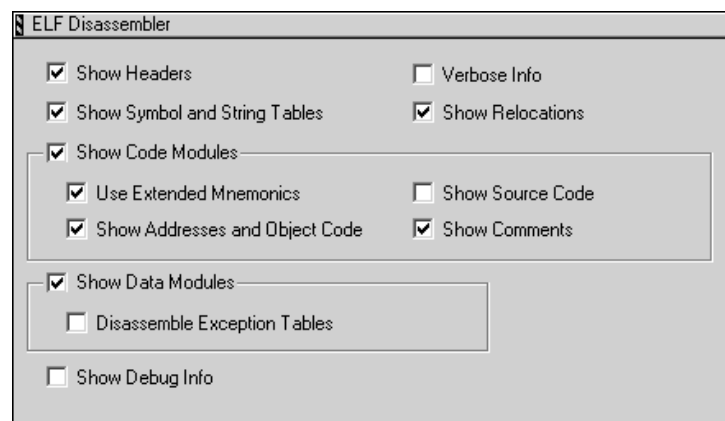
- .data.char — initialized static or global character data objects
- .bss.char — uninitialized static or global character data objects
- .const.data.char — const qualified character objects and static string data

You can locate these data sections in the lower half of the memory map, making sure that the data can be addressed.

## ELF Disassembler

Use the ELF Disassembler panel (Figure 4.9) to specify the content and display format for disassembled object files. Table 4.10 explains the elements of this panel. (To view a disassembled module, select **Project>Disassemble** from the main-window menu bar.)

**Figure 4.9 ELF Disassembler Panel**



# Freescall Semiconductor, Inc.

## Target Settings

DSP56800E-Specific Target Settings Panels

**Table 4.10 ELF Disassembler Panel Elements**

Element	Purpose	Comments
Show Headers checkbox	Checked — Disassembled output includes ELF header information. Clear — Disassembled output does not include this information.	
Show Symbol and String Tables checkbox	Checked — Disassembled modules include symbol and string tables. Clear — Disassembled modules do not include these tables.	
Verbose Info checkbox	Checked — ELF file includes additional information. Clear — ELF file does not include additional information.	For the .symtab section, additional information includes numeric equivalents for some descriptive constants. For the .line and .debug sections, additional information includes an unstructured hex dump.
Show Relocations checkbox	Checked — Shows relocation information for corresponding text (.rela.text) or data (.rela.data) section. Clear — Does not show relocation information.	
Show Code Modules checkbox	Checked — Disassembler outputs ELF code sections for the disassembled module. Enables subordinate checkboxes. Clear — Disassembler does not output these sections. Disables subordinate checkboxes.	Subordinate checkboxes are Use Extended Mnemonics, Show Addresses and Object Code, Show Source Code, and Show Comments.
Use Extended Mnemonics checkbox	Checked — Disassembler lists extended mnemonics for each instruction of the disassembled module. Clear — Disassembler does not list extended mnemonics.	This checkbox is available only if the Show Code Modules checkbox is checked.
Show Addresses and Object Code checkbox	Checked — Disassembler lists address and object code for the disassembled module. Clear — Disassembler does not list this code.	This checkbox is available only if the Show Code Modules checkbox is checked.
Show Source Code checkbox	Checked — Disassembler lists source code for the current module. Clear — Disassembler does not list source code.	Source code appears in mixed mode, with line-number information from the original C source file. This checkbox is available only if the Show Code Modules checkbox is checked.

**Table 4.10 ELF Disassembler Panel Elements (*continued*)**

Element	Purpose	Comments
Show Comments checkbox	Checked — Disassembler comments appear in sections that have comment columns. Clear — Disassembler does not produce comments.	This checkbox is available only if the Show Code Modules checkbox is checked.
Show Data Modules checkbox	Checked — Disassembler outputs ELF data sections, such as .data and .bss, for the disassembled module. Clear — Disassembler does not output ELF data sections.	
Disassemble Exception Table checkbox	<b>Leave clear.</b>	Does not apply to C.
Show Debug Info checkbox	Checked — Disassembler includes DWARF symbol information in output. Clear — Disassembler does not include this information in output.	

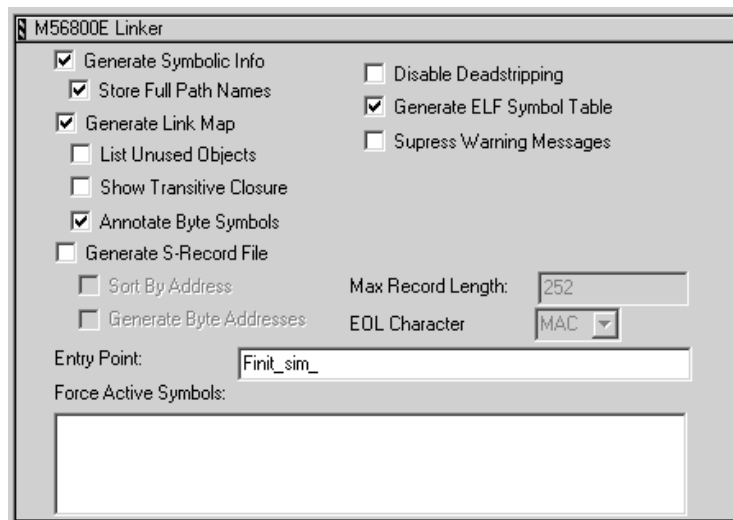
## M56800E Linker

Use the M56800E Linker panel (Figure 4.10) to specify linker behavior of the linker. (This panel is only available if the current build target uses the M56800E Linker.) Table 4.11 explains the elements of this panel.

## Target Settings

### DSP56800E-Specific Target Settings Panels

**Figure 4.10 M56800E Linker Panel**



**Table 4.11 M56800E Linker Panel Elements**

Element	Purpose	Comments
Generate Symbolic Info checkbox	Checked — Linker generates debugging information, within the linked ELF file. Clear — Linker does not generate debugging information.	If you select Project>Debug from the main-window menu bar, the IDE automatically enables this option. Clearing this checkbox prevents you from using the CodeWarrior debugger on your project; it also disables the subordinate Store Full Path Names checkbox.
Store Full Path Names checkbox	Checked — Linker includes full path names for source files. (Default) Clear — Linker uses only file names.	This checkbox is available only if the Generate Symbolic Info checkbox is checked.
Generate Link Map checkbox	Checked — Linker generates a link map. Enables subordinate checkboxes List Unused Objects, Show Transitive Closure, and Annotate Byte Symbols. Clear — Linker does not generate a link map.	A link map shows which file provided the definition of each object and function, the address of each object and function, a memory map of section locations, and values of linker-generated symbols. It also lists unused but unstripped symbols.



**Table 4.11 M56800E Linker Panel Elements (*continued*)**

Element	Purpose	Comments
List Unused Objects checkbox	Checked — Linker includes unused objects in the link map. Clear — Linker does not include unused objects in the link map.	This checkbox is available only if the Generate Link Map checkbox is checked.
Show Transitive Closure checkbox	Checked — Link map includes a list of all objects that main( ) references. Clear — Link map does not include this list.	Text after this table includes an example list. This checkbox is available only if the Generate Link Map checkbox is checked.
Annotate Byte Symbols	Checked — Linker includes B annotation for byte data types (e.g., char) in the Linker Command File. Clear — By default, the Linker does not include the B annotation in the Linker Command File. Everything without the B annotation is a word address.	For an example of the Linker Command File with and without the B annotation, see Listing 4.3.
Disable Deadstripping checkbox	Checked — Prevents the linker from stripping unused code or data. Clear — Lets the linker deadstrip.	
Generate ELF Symbol Table checkbox	Checked — Linker includes and ELF symbol table and relocation list in the ELF executable file. Clear — Linker does not include these items in the ELF executable file.	
Suppress Warning Messages checkbox	Checked — Linker does not display warnings in the message window. Clear — Linker displays warnings in the message window.	
Generate S-Record File checkbox	Checked — Linker generates an output file in S-record format. Activates subordinate checkboxes. Clear — Linker does not generate an S-record file.	For the DSP56800E, this option outputs three S-record files: .s (both P and X memory contents), .p (P memory contents), and .x (X memory contents). The linker puts S-record files in the output folder (a sub-folder of the project folder.)
Sort By Address checkbox	Checked — Enables the compiler to use byte addresses to sort type S3 S-records that the linker generates. Clear — Does not enable byte-address sorting.	This checkbox is available only if the Generate S-Record File checkbox is checked.

# Freescale Semiconductor, Inc.

## Target Settings

### DSP56800E-Specific Target Settings Panels

**Table 4.11 M56800E Linker Panel Elements (continued)**

Element	Purpose	Comments
Generate Byte Addresses checkbox	Checked — Enables the linker to generate type S3 S-records in bytes. Clear — Does not enable byte generation.	This checkbox is available only if the Generate S-Record File checkbox is checked.
Max Record Length text box	Specifies the maximum length of type S3 S-records that the linker generates, up to 256 bytes.	The CodeWarrior debugger handles 256-byte S-records. If you use different software to load your embedded application, This text box should specify that software's maximum length for S-records. This checkbox is available only if the Generate S-Record File checkbox is checked.
EOL Character list box	Specifies the end-of-line character for the type S3 S-record file: Mac, DOS, or UNIX format.	This checkbox is available only if the Generate S-Record File checkbox is checked.
Entry Point text box	Specifies the program starting point — the first function the linker uses when the program runs.	Text after this table includes additional information about the entry point.
Force Active Symbols text box	Directs the linker to include symbols in the link, even if those symbols are not referenced. Makes symbols immune to deadstripping.	Separate multiple symbols with single spaces.

Check the Show Transitive Closure checkbox to have the link map include the list of objects main( ) references. Consider the sample code of Listing 4.1. If the Show Transitive Closure option is in effect and you compile this code, the linker generates a link map file that includes the list of Listing 4.2.

#### Listing 4.1 Sample Code for Show Transitive Closure

```
void foot( void ){ int a = 100; }
void pad( void ){ int b = 101; }

int main( void ){
    foot();
    pad();
    return 1;
}
```

---

## Listing 4.2 Link Map File: List of main( ) references

---

```
# Link map of Finit_sim_
1] interrupt_vectors.text found in 56800E_vector.asm
2] sim_intRoutine (notype,local) found in 56800E_vector.asm
2] Finit_sim_ (func,global) found in 56800E_init.asm
3] Fmain (func,global) found in M56800E_main.c
4] Ffoot (func,global) found in M56800E_main.c
4] Fpad (func,global) found in M56800E_main.c
3] F__init_sections (func,global) found in Runtime 56800E.lib
initsections.o
4] Fmemset (func,global) found in MSL C 56800E.lib mem.o
5] F__fill_mem (func,global) found in MSL C 56800E.lib
mem_funcs.o
1] Finit_sim_ (func,global) found in 56800E_init.asm
```

---

Use the Entry Point text box to specify the starting point for a program. The default function this text box names is in the startup code that sets up the DSP56800E environment before your code executes. This function and its corresponding startup code depend on your stationery selection.

For hardware-targeted stationery, the startup code is on the path:

support\*<name of hardware, e.g., M56852E>*\startup

For simulator-targeted stationery, the startup code is on the path:

support\M56800E\init

The startup code performs such additional tasks as clearing the hardware stack, creating an interrupt table, and getting the addresses for the stack start and exception handler. The final task for the startup code is call your main( ) function.

Check the Annotate Byte Symbols checkbox to have the link map include the B annotation for byte addresses and no B annotation for word addresses (Listing 4.3).

---

## Listing 4.3 Example of Annotate Byte Symbols

---

```
int myint;
char mychar;

B 0000049C 00000001 .bss Fmychar (main.c)
0000024F 00000001 .bss Fmyint (main.c)
```

---

## Target Settings

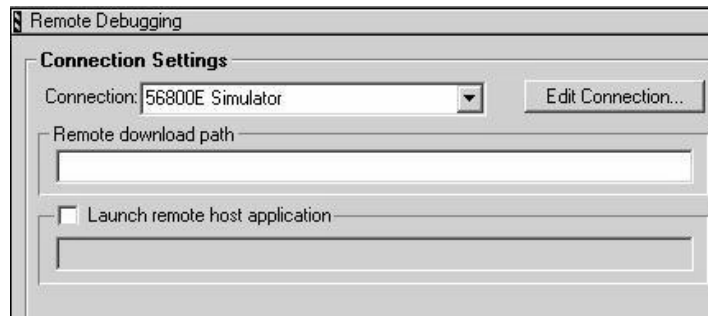
DSP56800E-Specific Target Settings Panels

## Remote Debugging

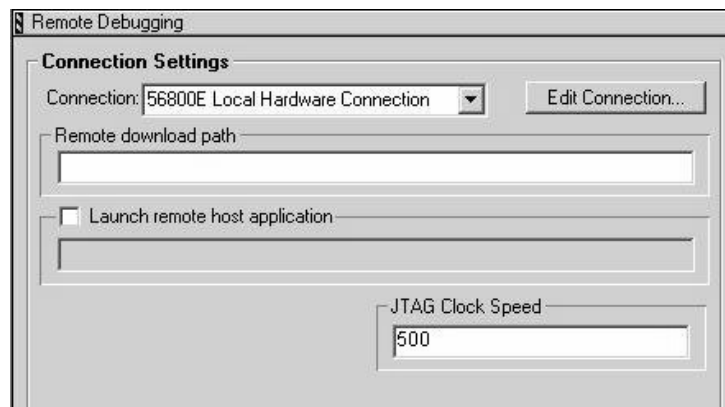
Use the Remote Debugging panel (Figure 4.11, Figure 4.12) to set parameters for communication between a DSP56800E board or Simulator and the CodeWarrior DSP56800E debugger. Table 4.12 explains the elements of this panel.

**NOTE** Communications specifications also involve settings of the debugging M56800E Target panel (Figure 4.13).

**Figure 4.11 Remote Debugging Panel (56800E Simulator)**



**Figure 4.12 Remote Debugging Panel (56800E Local Connection)**



**Table 4.12 Remote Debugging Panel Elements**

Element	Purpose	Comments
Connection list box	Specifies the connection type: <ul style="list-style-type: none"> <li>56800E Simulator — appropriate for testing code on the simulator before downloading code to an actual board.</li> <li>56800E Local Hardware Connection (CSS) — appropriate for using your computer's command converter server, connected to a DSP56800E board.</li> </ul>	Selecting 56800E Simulator keeps the panel as Figure 4.11 shows. Selecting Local Hardware Connection adds the JTAG Clock Speed text box to the panel, as Figure 4.12 shows.
Remote Download Path text box		Not supported at this time.
Launch Remote Host Application checkbox		Not supported at this time.
JTAG Clock Speed text box	Specifies the JTAG lock speed for local hardware connection. (Default is 500 kilohertz.)	This list box is available only if the Connection list box specifies Local Hardware Connection (CSS). The HTI will not work properly with a clock speed over 500 kHz.

## M56800E Target (Debugging)

Use the debugging M56800E Target panel (Figure 4.13) to set parameters for communication between a DSP56800E board or Simulator and the CodeWarrior DSP56800E debugger. Table 4.12 explains the elements of this panel.

---

**NOTE** Communications specifications also involve settings of the Remote Debugging panel (Figure 4.11, Figure 4.12).

---

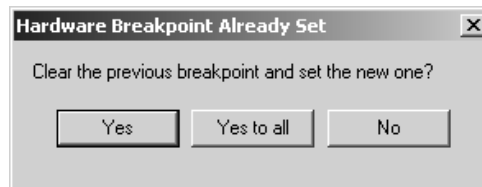
## Auto-clear previous breakpoint on new breakpoint request

This option is only available when you enable the **Use hardware breakpoints** option. When you also enable the **Auto-clear previous hardware breakpoint** and set a breakpoint, the original breakpoint is automatically cleared and the new breakpoint is

## Target Settings

### DSP56800E-Specific Target Settings Panels

immediately set. If you disable the **Auto-clear previous hardware breakpoint** option and attempt to set another breakpoint, you will be prompted the following message:

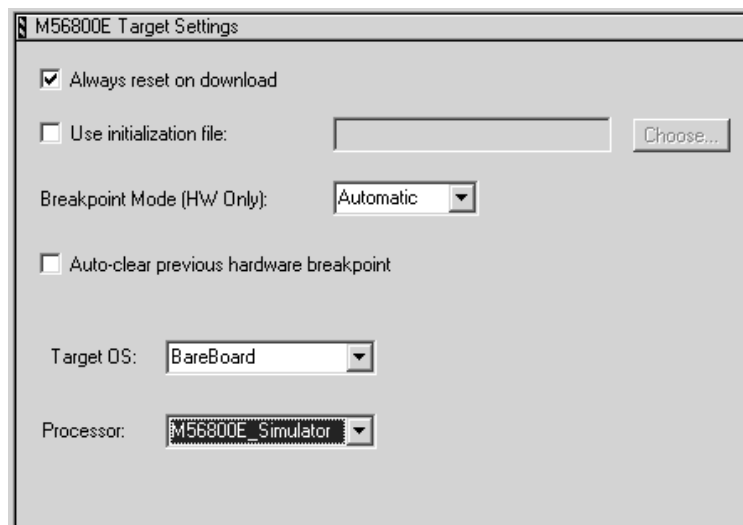


If you click the **Yes** button, the previous breakpoint is cleared and the new breakpoint is set.

If you click the **Yes to all** button, the **Auto-clear previous hardware breakpoint** option is enabled and the previously set breakpoint is cleared out without prompting for every subsequent occurrence.

If you click the **No** button, the previous breakpoint is kept and the new breakpoint request is ignored.

**Figure 4.13 Debugging M56800E Target Panel**



**Table 4.13 Debugging M56800E Target Panel Elements**

Element	Purpose	Comments
Always reset on download checkbox	Checked — IDE issues a reset to the target board each time you connect to the board. Clear — IDE does not issue a reset each time you connect to the target board.	
Use initialization file checkbox	Checked — After a reset, the IDE uses an optional hardware initialization file before downloading code. Clear — IDE does not use a hardware initialization file.	The Use initialization file text box specifies the file. Text immediately after this table gives more information about initialization files.
Use initialization file text box	Specifies the initialization file.	Applicable only if the Use initialization file checkbox is checked.
BreakpointMode checkbox	Specifies the breakpoint mode: <ul style="list-style-type: none"> <li>Automatic — CodeWarrior software determines when to use software or hardware breakpoints.</li> <li>Software — CodeWarrior software always uses software breakpoints.</li> <li>Hardware — CodeWarrior software always uses the available hardware breakpoints.</li> </ul>	Software breakpoints contain debug instructions that the debugger writes into your code. You cannot set such breakpoints in flash, as it is read-only. Hardware breakpoints use the on-chip debugging capabilities of the DSP56800E. The number of available hardware breakpoints limits these capabilities. Note, Breakpoint Mode only effects HW targets.
Auto-clear previous hardware breakpoint	Checked — Automatically clears the previous hardware breakpoint. Clear — Does not Automatically clears the previous hardware breakpoint.	
Target OS list box	Specifies the OS	Selects the OS plug-in. The <b>BareBoard</b> option does not use an OS plug-in.
Processor list box	Specifies the processor	Currently this selects the register layout.

An initialization file consists of text instructions telling the debugger how to initialize the hardware after reset, but before downloading code. You can use initialization file commands to assign values to registers and memory locations, and to set up flash memory parameters.

The initialization files of your IDE are on the path:

{CodeWarrior path}\M56800E Support\initialization

# Freescale Semiconductor, Inc.

## Target Settings

### DSP56800E-Specific Target Settings Panels

The name of each initialization file includes the number of the corresponding processor, such as 568345. Each file with “\_ext” enables the processor’s external memory. If the processor has Flash memory, the initialization file with “\_flash” enables both Flash and external memory.

To set up an initialization file:

1. In the debugging M56800E Target panel, check the Use initialization file checkbox.
2. Specify the name of the initialization file, per either substep a or b:
  - a. Type the name in the Use initialization file text box. If the name is not a full pathname, the debugger searches for the file in the project directory. If the file is not in this directory, the debugger searches on the path:  
`{CodeWarrior path}\M56800E Support\ initialization directory.`
  - b. Click the **Choose** button; the Choose file dialog box appears. Navigate to the appropriate file. When you select the file, the system puts its name in the Use initialization file text box.

Each text line of a command file begins with a command or the comment symbol #. The system ignores comment lines, as well as blank lines.

Table 4.14 lists the supported commands and their arguments. For a more detailed description of the Flash Memory commands see “Flash Memory Commands.”

**Table 4.14 Initialization File Commands and Arguments**

Command	Arguments	Description
writemem	<addr> <value>	Writes a 16-bit value to the specified P: Memory location.
writexmem	<addr> <value>	Writes a 16-bit value to the specified X: Memory location.
writereg	<regName> <value>	Writes a 16-bit value to the specified register.
set_hfmckld	<value>	Writes the flash memory’s clock divider value to the hfmckld register



**Table 4.14 Initialization File Commands and Arguments (continued)**

Command	Arguments	Description
set_hfm_base	<address>	Sets the address of hfm_base. This is the map location of the flash memory control registers in X: Memory.
add_hfm_unit	<startAddr><endAddr> <bank><numSectors> <pageSize><progMem> <boot><interleaved>	Adds a flash memory unit to the list and sets its parameter values.
set_hfm_programmer_base	<address>	Specifies the address where the onboard flash programmer will be loaded in P: Memory.
set_hfm_prog_buffer_base	<address>	Specifies where the data to be programmed will be loaded in X: Memory.
set_hfm_prog_buffer_size	<size>	Specifies the size of the buffer in X: Memory which will hold the data to be programmed.
set_hfm_erase_mode	<units   pages   all>	Sets the erase mode.
set_hfm_verify_erase	<1   0>	Sets the flash memory erase verification mode.
set_hfm_verify_program	<1   0>	Sets the flash program verification mode.
unlock_flash_on_connect	<1   0>	Unlocks and erases flash memory immediately upon connection.

## Remote Debug Options

Use the Remote Debug Options panel (Figure 4.14) to specify different remote debug options.

Target Settings  
DSP56800E-Specific Target Settings Panels

Figure 4.14 Remote Debug Options

Remote Debug Options

Program Download Options

Section Type	Initial Launch		Successive Runs	
	Download	Verify	Download	Verify
Executable	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Constant Data	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Initialized Data	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Uninitialized Data	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>

Memory Configuration Options

☐ Use Memory Configuration File

**Table 4.15 Remote Debug Options Panel Elements**

Element	Purpose	Comments
Program Download Options area	Checked Download checkboxes specify the section types to be downloaded on initial launch and on successive runs. Checked Verify checkboxes specify the section types to be verified (that is, read back to the linker).	<p>Section types:</p> <ul style="list-style-type: none"> <li>• Executable — program-code sections that have X flags in the linker command file.</li> <li>• Constant Data — program-data sections that do not have X or W flags in the linker command file.</li> <li>• Initialized Data — program-data sections with initial values. These sections have W flags, but not X flags, in the linker command file.</li> <li>• Uninitialized Data — program-data sections without initial values. These sections have W flags, but not X flags, in the linker command file.</li> </ul>
Use Memory Configuration File checkbox		Not supported at this time.

# Freescale Semiconductor, Inc.

## Target Settings

*DSP56800E-Specific Target Settings Panels*

---

# Processor Expert Interface

---

Your CodeWarrior™ IDE features a Processor Expert™ plug-in interface, for rapid development of embedded applications. This chapter explains Processor Expert concepts, and Processor Expert additions to the CodeWarrior visual interface. This chapter includes a brief tutorial exercise.

This chapter contains these sections:

- Processor Expert Overview
- Processor Expert Windows
- Processor Expert Tutorial

## Processor Expert Overview

The Processor Expert Interface (PEI) is an integrated development environment for applications based on DSP56800/E (or many other) embedded microcontrollers. It reduces development time and cost for applications. Its code makes very efficient use of microcontroller and peripheral capabilities. Furthermore, it helps develop code that is highly portable.

Features include:

- **Embedded Beans™ components** — Each bean encapsulates a basic functionality of embedded systems, such as CPU core, CPU on-chip peripherals, and virtual devices. To create an application, you select, modify, and combine the appropriate beans.
  - The **Bean Selector** window lists all available beans, in an expandable tree structure. The Bean Selector describes each bean; some descriptions are extensive.
  - The **Bean Inspector** window lets you modify bean properties, methods, events, and comments.
- **Processor Expert page** — This additional page for the CodeWarrior project window lists project CPUs, beans, and modules, in a tree structure. Selecting or double-clicking items of the page opens or changes the contents of related Processor Expert windows.

- **Target CPU window** — This window depicts the target microprocessor as a simple package or a package with peripherals. As you move the cursor over this picture's pins, the window shows pin numbers and signals. Additionally, you can have this window show a scrollable block diagram of the microprocessor.
- **CPU Structure window** — This window shows the relationships of all target-microprocessor elements, in an expandable-tree representation.
- **CPU Types Overview** — This reference window lists all CPUs that your Processor Expert version supports.
- **Memory Map** — This window shows the CPU address space, plus mapping for internal and external memory.
- **Resource Meter** — This window shows the resource allocation for the target microprocessor.
- **Peripheral Usage Inspector** — This window shows which bean allocates each on-chip peripheral.
- **Installed Beans Overview** — This reference window provides information about all installed beans in your Processor Expert version.
- **Driver generation** — The PEI suggests, connects, and generates driver code for embedded-system hardware, peripherals, and algorithms.
- **Top-Down Design** — A developer starts design by defining application behavior, rather than by focussing on how the microcontroller works.
- **Extensible beans library** — This library supports many microprocessors, peripherals, and virtual devices.
- **Beans wizard** — This external tool helps developers create their own custom beans.
- **Extensive help information** — You access this information either by selecting Help from the Program Expert menu, or by clicking the Help button of any Processor Expert window or dialog box.

## Processor Expert Code Generation

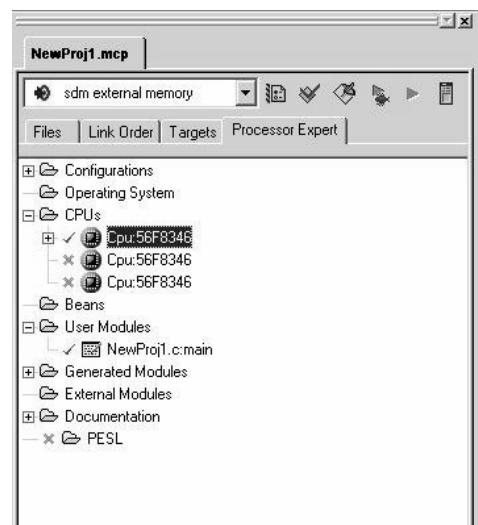
The PEI manages your CPU and other hardware resources, so that you can concentrate on virtual prototyping and design. Your steps for application development are:

1. Creating a CodeWarrior project, specifying the Processor Expert stationery appropriate for your target processor.
2. Configuring the appropriate CPU bean.
3. Selecting and configuring the appropriate function beans.

4. Starting code design (that is, building the application).

As you create the project, the project window opens in the IDE main window. This project window has a Processor Expert page (Figure 5.1). The Processor Expert Target CPU window also appears at this time. So does the Processor Expert bean selector window, although it is behind the Target CPU window.

**Figure 5.1 Project Window: Processor Expert Page**



When you start code design, the PEI generates commented code from the bean settings. This code generation takes advantage of the Processor Expert CPU knowledge system and solution bank, which consists of hand-written, tested code optimized for efficiency.

To add new functionalities, you select and configure additional beans, then restart code design. Another straightforward expansion of PEI code is combining other code that you already had produced with different tools.

## Processor Expert Beans

Beans encapsulate the most-required functionalities for embedded applications. Examples include port bit operations, interrupts, communication timers, and A/D converters.

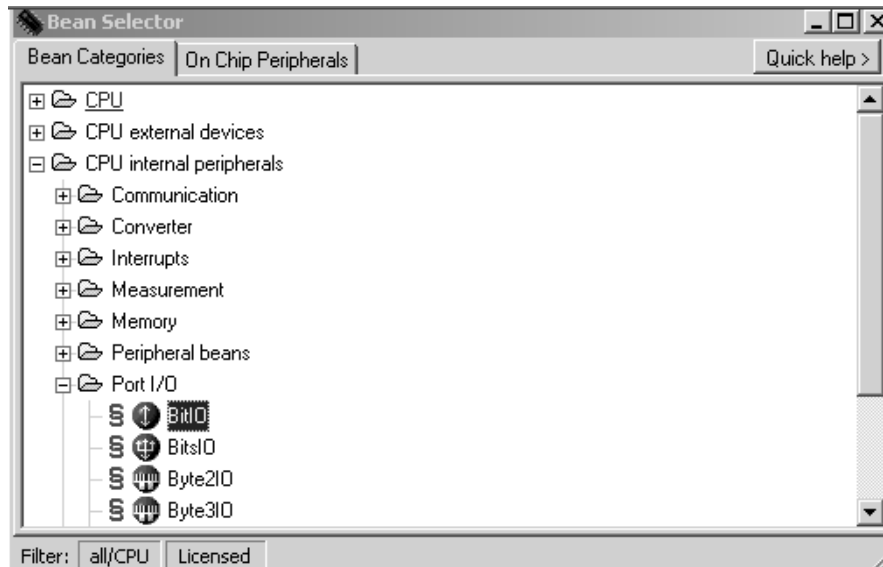
The Bean Selector (Figure 5.2) helps you find appropriate beans by category: processor, MCU external devices, MCU internal peripherals, or on-chip peripherals.

## Processor Expert Interface

### Processor Expert Overview

To open the bean selector, select **Processor Expert > View > Bean Selector**, from the main-window menu bar.

**Figure 5.2 Bean Selector**

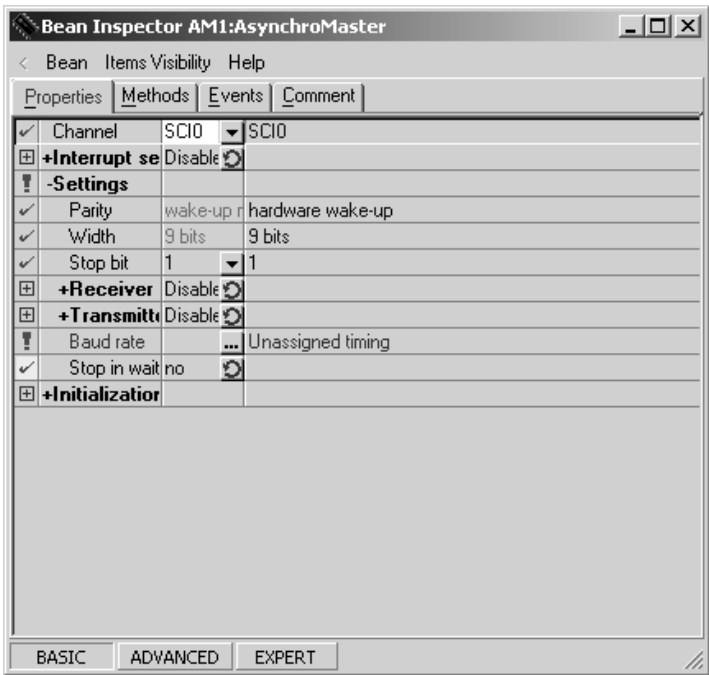


The bean selector's tree structures list all available beans; double-clicking the name adds the bean to your project. Clicking the Quick Help button opens or closes an explanation pane that describes the highlighted bean.

Once you determine the appropriate beans, you use the Bean Inspector (Figure 5.3) to fine tune each bean, making it optimal for your application.



Figure 5.3 Bean Inspector



Using the Bean Inspector to set a bean’s initialization properties automatically adds bean initialization code to CPU initialization code. You use the Bean Inspector to adjust bean properties, so that generated code is optimal for your application.

Beans greatly facilitate management of on-chip peripherals. When you choose a peripheral from bean properties, the PEI presents all possible candidates. But the PEI indicates which candidates already are allocated, and which are not compatible with current bean settings.

## Processor Expert Menu

Table 5.1 explains the selections of the Processor Expert menu.

# Freescale Semiconductor, Inc.

## Processor Expert Interface Processor Expert Overview

**Table 5.1 Processor Expert Menu Selections**

Item	Subitem	Action
Open Processor Expert	none	Opens the PEI for the current project. (Available only if the current project does not already involve the PEI.)
Code Design <Project>	none	Generates code, including drivers, for the current project. Access these files via the Generate Code folder, of the project-window Files page.
Undo Last Code Design	none	Deletes the most recently-generated code, returning project files to their state after the previous, error-free code generation.
View	Project Panel	Brings the Processor Expert page to the front of the CodeWarrior project window. (Not available if the project window does not include a Processor Expert page.)
	Bean Inspector	Opens the Bean Inspector window, which gives you access to bean properties.
	Bean Selector	Opens the Beans Selector window, which you use to choose the most appropriate beans.
	Target CPU Package	Opens the Target CPU Package window, which depicts the processor. As you move your cursor over the pins of this picture, text boxes show pin numbers and signal names.
	Target CPU Block Diagram	Opens the Target CPU Package window, but portrays the processor as a large block diagram. Scroll bars let you view any part of the diagram. As you move your cursor over modules, floating text boxes identify pin numbers and signals.
	Error Window	Opens the Error Window, which shows hints, warnings, and error messages.
	Resource Meter	Opens the Resource Meter window, which shows usage and availability of processor resources.
View (continued)	Target CPU Structure	Opens the CPU Structure window, which uses an expandible tree structure to portray the processor.

**Table 5.1 Processor Expert Menu Selections (*continued*)**

Item	Subitem	Action
	Peripherals Usage Inspector	Opens the Peripherals Usage Inspector window, which shows which bean allocates each peripheral.
	Peripheral Initialization Inspector	Opens the Peripherals Initialization Inspector window, which show the initialization value and value after reset for all peripheral register bits.
	Installed Beans Overview	Opens the Beans Overview window, which provides information about all beans in your project.
	CPU Types Overview	Opens the CPU Overview window, which lists supported processors in an expandable tree structure.
	CPU Parameters Overview	Opens the CPU Parameters window, which lists clock-speed ranges, number of pins, number of timers, and other reference information for the supported processors.
	Memory Map	Opens the Memory Map window, which depicts CPU address space, internal memory, and external memory.
Tools	<tool name>	Starts the specified compiler, linker or other tool. (You use the Tools Setup window to add tool names to this menu.)
	SHELL	Opens a command-line window.
	Tools Setup	Opens the Tools Setup window, which you use to add tools to this menu.
Help	Processor Expert Help	Opens the help start page.
	Introduction	Opens the PEI help introduction.
	Benefits	Opens an explanation of PEI benefits.
	User Interface	Opens an explanation of the PEI environment.
	Tutorial	[None available for the DSP56800/E.]
	Quick Start	Opens PEI quick start instructions.
Help (continued)	Embedded Beans	Opens the first page of a description database of all beans.

# Freescale Semiconductor, Inc.

## Processor Expert Interface Processor Expert Overview

**Table 5.1 Processor Expert Menu Selections (*continued*)**

Item	Subitem	Action
	Embedded Beans Categories	Opens the first page of a description database of beans, organized by category.
	Supported CPUs, Compilers, and Debuggers	Opens the list of processors and tools that the PEI plug-in supports.
	PESL Library User Manual	Opens the Processor Expert System Library, for advanced developers.
	User Guide	Opens a .pdf guide that focuses on the DSP56800/E processor family.
	Search in PDF Documentation of the Target CPU	Opens documentation of the target processor, in a .pdf search window.
	Go to Processor Expert Home Page	Opens your default browser, taking you to the PEI home page.
	About Processor Expert	Opens a standard About dialog box for the PEI.
Update	Update Processor Expert Beans from Package	Opens the Open Update Package window. You can use this file-selection window to add updated or new beans (which you downloaded over the web) to your project.
	Check Processor Expert Web for updates	Checks for updates available over the web. If any are available, opens your default browser, so that you can download them.
Bring PE Windows to Front	none	Moves PEI windows to the front of your monitor screen.
Arrange PE Windows	none	Restores the default arrangement of all open PEI windows.

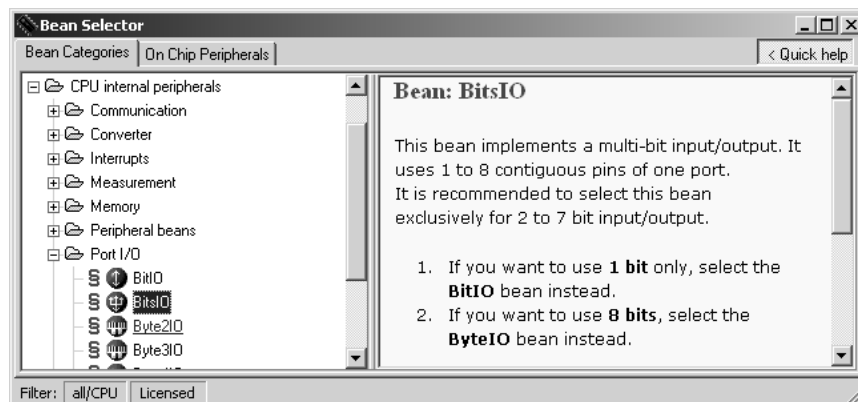
## Processor Expert Windows

This section illustrates important Processor Expert windows and dialog boxes.

### Bean Selector

The **Bean Selector** window (Figure 5.4) explains which beans are available, helping you identify those most appropriate for your application project. To open this window, select **Processor Expert > View > Bean Selector**, from the main-window menu bar.

Figure 5.4 Bean Selector Window



The **Bean Categories** page, at the left side of this window, lists the available beans in category order, in an expandable tree structure. Green string bean symbols identify beans that have available licenses. Grey string bean symbols identify beans that do not have available licenses.

The **On-Chip Peripherals** page lists beans available for specific peripherals, also in an expandable tree structure. Yellow folder symbols identify peripherals fully available. Light blue folder symbols identify partially used peripherals. Dark blue folder symbols identify fully used peripherals.

Bean names are black; bean template names are blue. Double-click a bean name to add it to your project.

Click the Quick Help button to add the explanation pane to the right side of the window, as Figure 5.4 shows. This pane describes the selected (highlighted) bean. Use the scroll bars to read descriptions that are long.

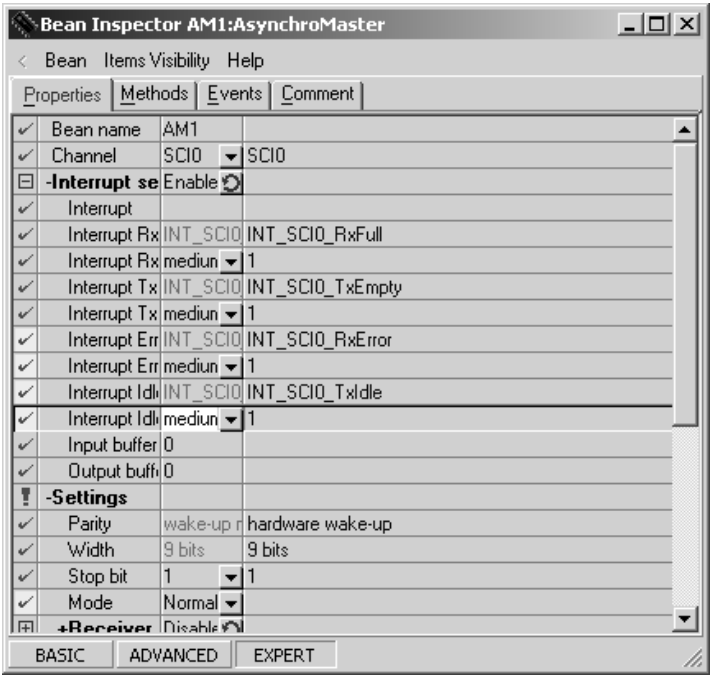
Processor Expert Interface  
Processor Expert Windows

Click the two buttons at the bottom of the window to activate or deactivate filters. If the **all/CPU** filter is active, the window lists only the beans for the target CPU. If the license filter is active, the window lists only the beans for which licenses are available.

Bean Inspector

The **Bean Inspector** window (Figure 5.5) lets you modify bean properties and other settings. To open this window, select **Processor Expert > View > Bean Inspector**, from the main-window menu bar.

Figure 5.5 Bean Inspector Window



This window shows information about the currently selected bean — that is, the highlighted bean name in the project-window Processor Expert page. The title of the Bean Inspector window includes the bean name.

The Bean Inspector consists of Properties, Methods, Events, and Comment pages. The first three pages have these columns:

- **Item names** — Items to be set. Double-click on group names to expand or collapse this list. For the Method or Event page, double-clicking on an item may open the file editor, at the corresponding code location.
- **Selected settings** — Possible settings for your application. To change any ON/OFF-type setting, click the circular-arrow button. Settings with multiple possible values have triangle symbols: click the triangle to open a context menu, then select the appropriate value. Timing settings have an ellipsis (...) button: click this button to open a setting dialog box.
- **Setting status** — Current settings or error statuses.

Use the comments page to write any notations or comments you wish.

---

<b>NOTE</b>	<p>If you have specified a target compiler, the Bean Inspector includes an additional Build options page for the CPU bean.</p> <p>If your project includes external peripherals, the Bean Inspector includes an additional Used page. Clicking a circular-arrow button reserves a resource for connection to an external device. Clicking the same button again frees the resource.</p>
-------------	---

---

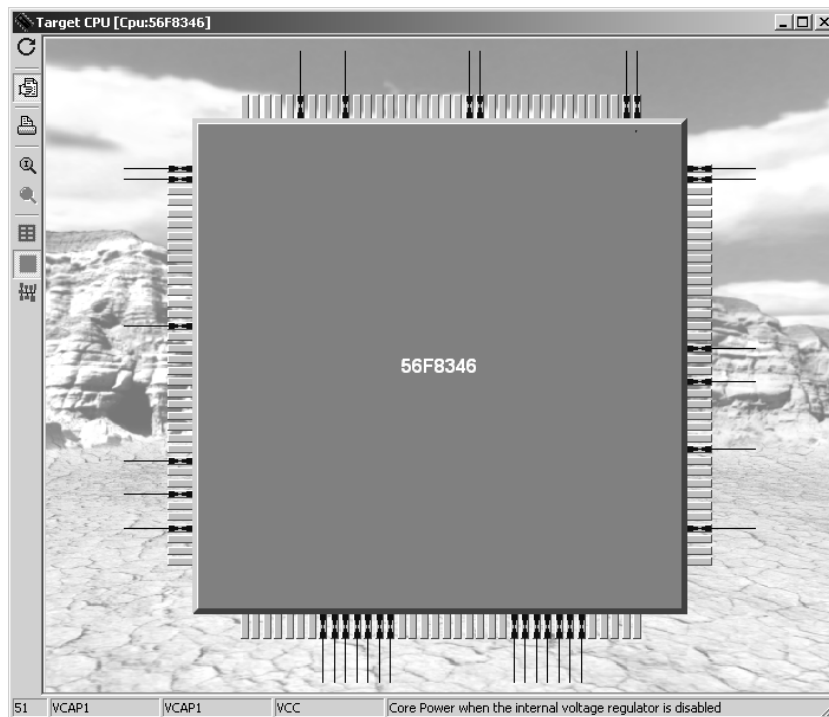
The Basic, Advanced, and Expert view mode buttons, at the bottom of the window, let you change the detail level of Bean Inspector information.

The Bean Inspector window has its own menu bar. Selections include restoring default settings, saving the selected bean as a template, changing the bean's icon, disconnecting from the CPU, and several kinds of help information.

## Target CPU Window

The **Target CPU** window (Figure 5.6) depicts the target processor as a realistic CPU package, as a CPU package with peripherals, or as a block diagram. To open this window, select **Processor Expert > View > Target CPU Package**, from the main-window menu bar. (To have this window show the block diagram, you may select **Processor Expert > View > Target CPU Block Diagram**, from the main-window menu bar.)

**Figure 5.6 Target CPU Window: Package**

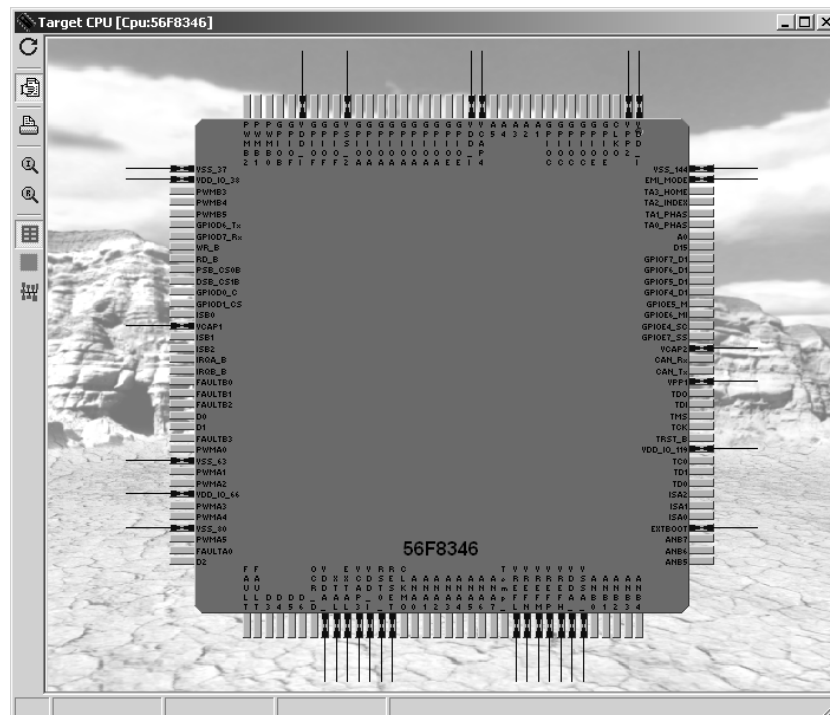


Arrows on pins indicate input, output, or bidirectional signals. As you move your cursor over the processor pins, text boxes at the bottom of this window show the pin numbers and signal names.



Use the control buttons at the left edge of this window to modify the depiction of the processor. One button, for example, changes the picture view the CPU package with peripherals. However, as Figure 5.7 shows, it is not always possible for the picture of a sophisticated processor to display internal peripherals.

**Figure 5.7 Target CPU Window: Package and Peripherals**



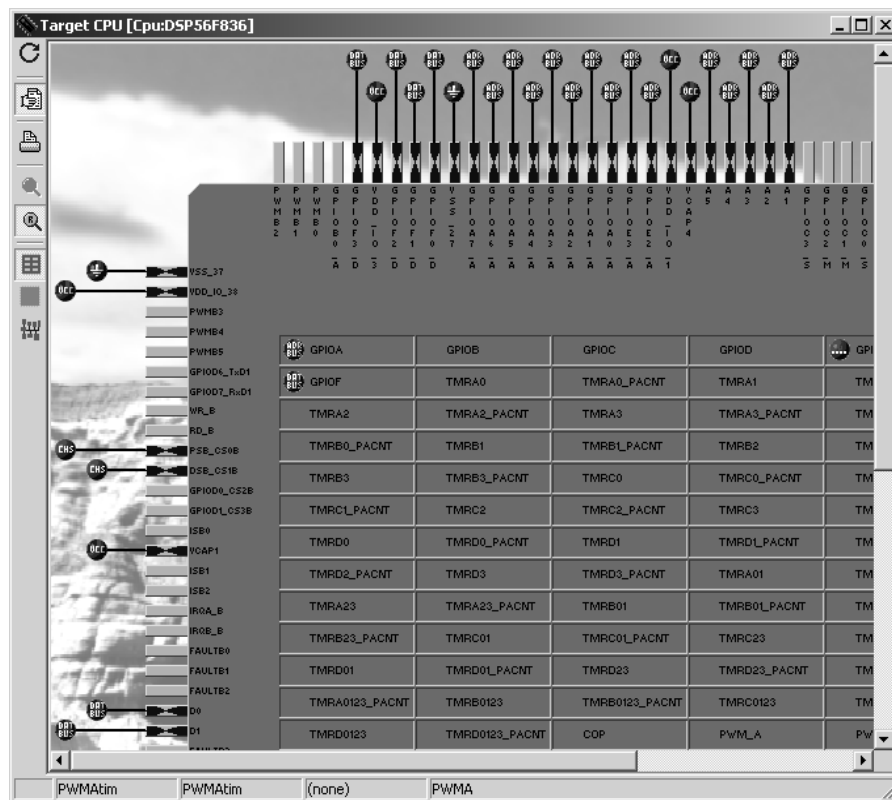
**Freescale Semiconductor, Inc.**

## Processor Expert Interface

### *Processor Expert Windows*

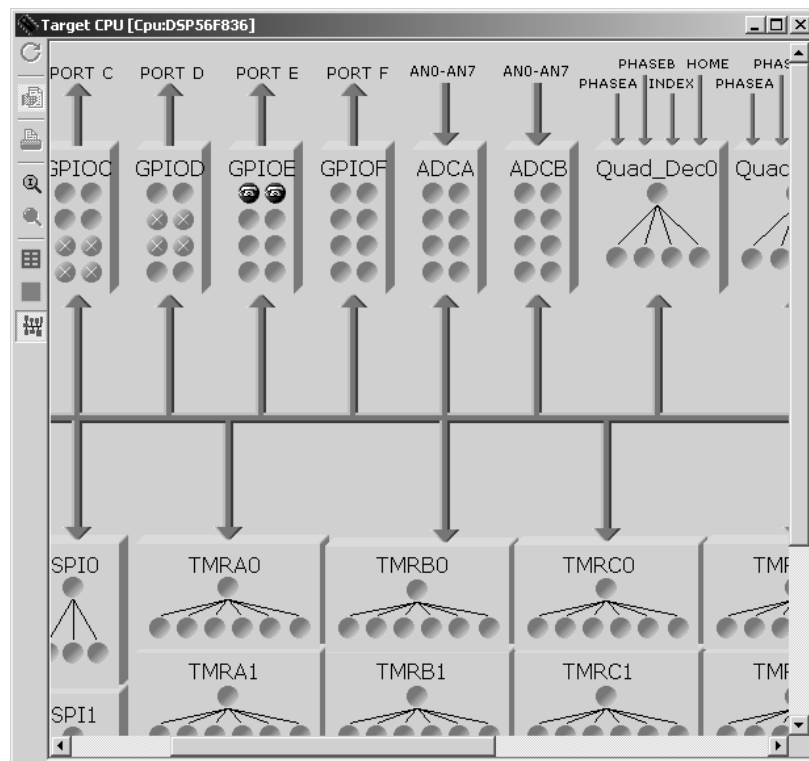
In such a case, you can click the **Always show internal peripheral devices** control button. Figure 5.8 shows that this expands the picture size, as necessary, to allow the peripheral representations. This view also includes bean icons (blue circles) attached to the appropriate processor pins. Use the scroll bars to view other parts of the processor picture.

**Figure 5.8 Target CPU Window: Peripherals and Bean Icons**



Click the Show MCU Block Diagram to change the picture to a block diagram, as Figure 5.9 shows. Use the scroll bars to view other parts of the diagram. (You can bring up the block diagram as you open the Target CPU window, by selecting **Processor Expert > View > Target CPU Block Diagram**, from the main-window menu bar.)

**Figure 5.9 Target CPU Window: Block Diagram**



Other control buttons at the left edge of the window let you:

- Show bean icons attached to processor pins.
- Rotate the CPU picture clockwise 90 degrees.
- Toggle default and user-defined names of pins and peripherals.
- Print the CPU picture.

## Processor Expert Interface

### Processor Expert Windows

---

---

<b>NOTE</b>	As you move your cursor over bean icons, peripherals, and modules, text boxes or floating hints show information such as names, descriptions, and the allocating beans.
-------------	---

---

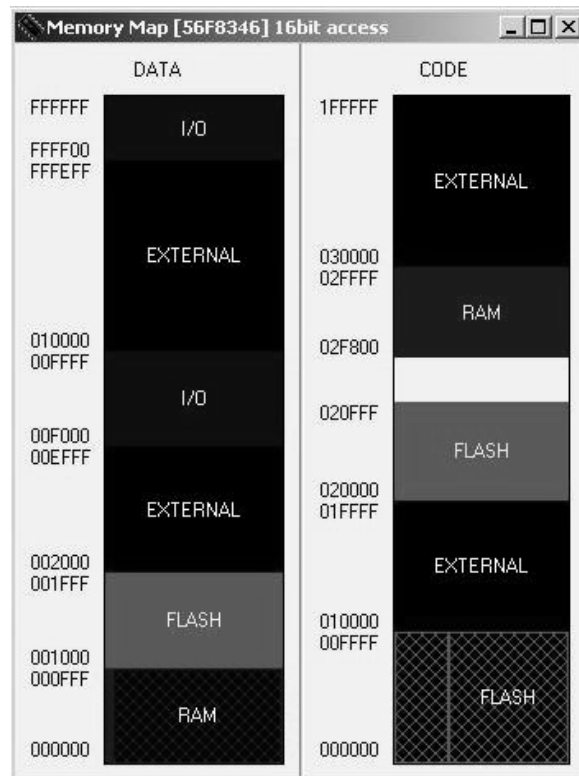
And note these additional mouse control actions for the Target CPU window:

- Clicking a bean icon selects the bean in the project window's Processor Expert page.
- Double-clicking a bean icon open the Bean Inspector, displaying information for that bean.
- Right-clicking a bean icon, a pin, or a peripheral opens the corresponding context menu.
- Double-clicking an ellipsis (...) bean icon opens a context menu of all beans using parts of the peripheral. Selecting one bean from this menu opens the Bean Inspector.
- Right-clicking an ellipsis (...) bean icon opens a context menu of all beans using parts of the peripheral. Selecting one bean from this menu opens the bean context menu.

## Memory Map Window

The **Memory Map** window (Figure 5.10) depicts CPU address space, and the map of internal and external memory. To open this window, select **Processor Expert > View > Memory Map**, from the main-window menu bar.

Figure 5.10 Memory Map Window



The color key for memory blocks is:

- White — Non-usable space
- Dark Blue — I/O space
- Medium Blue — RAM
- Light Blue — ROM

## Processor Expert Interface

### Processor Expert Windows

---

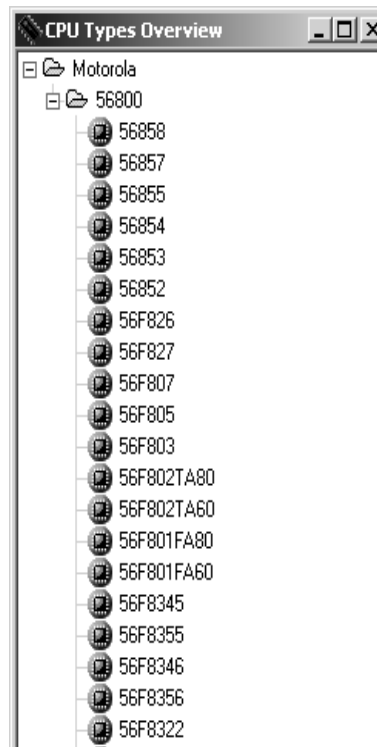
- Cyan — FLASH memory or EEPROM
- Black — External memory.

Pause your cursor over any block of the map to bring up a brief description.

## CPU Types Overview

The **CPU Types Overview** window (Figure 5.11) lists supported processors, in an expandable tree structure. To open this window, select **Processor Expert > View > CPU Types Overview**, from the main-window menu bar.

**Figure 5.11 CPU Types Overview Window**

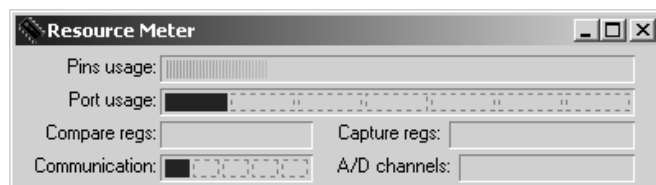


Right-click the window to open a context menu that lets you add the selected CPU to the project, expand the tree structure, collapse the tree structure, or get help information.

## Resource Meter

The **Resource Meter** window (Figure 5.12) shows the usage or availability of processor resources. To open this window, select **Processor Expert > View > Resource Meter**, from the main-window menu bar.

**Figure 5.12 Resource Meter Window**



Bars of this window indicate:

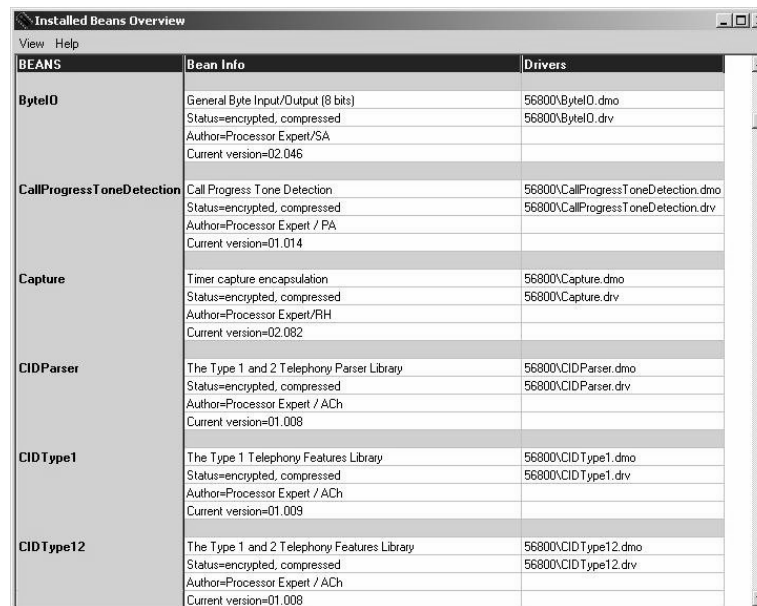
- The number of pins used
- The number of ports used
- Allocation of timer compare registers
- The number of timer capture registers used
- Allocation of serial communication channels
- Allocation of A/D converter channels.

Pausing your cursor over some fields of this window brings up details of specific resources.

## Installed Beans Overview

The **Installed Beans Overview** window (Figure 5.13) shows reference information about the installed beans. To open this window, select **Processor Expert > View > Installed Beans Overview**, from the main-window menu bar.

Figure 5.13 Installed Beans Overview Window



BEANS	Bean Info	Drivers
<b>ByteIO</b>	General Byte Input/Output (8 bits)	56800\ByteIO.dmo
	Status=encrypted, compressed	56800\ByteIO.drv
	Author=Processor Expert/SA	
	Current version=02.045	
<b>CallProgressToneDetection</b>	Call Progress Tone Detection	56800\CallProgressToneDetection.dmo
	Status=encrypted, compressed	56800\CallProgressToneDetection.drv
	Author=Processor Expert / PA	
	Current version=01.014	
<b>Capture</b>	Timer capture encapsulation	56800\Capture.dmo
	Status=encrypted, compressed	56800\Capture.drv
	Author=Processor Expert/RH	
	Current version=02.082	
<b>CIDParser</b>	The Type 1 and 2 Telephony Parser Library	56800\CIDParser.dmo
	Status=encrypted, compressed	56800\CIDParser.drv
	Author=Processor Expert / ACh	
	Current version=01.008	
<b>CIDType1</b>	The Type 1 Telephony Features Library	56800\CIDType1.dmo
	Status=encrypted, compressed	56800\CIDType1.drv
	Author=Processor Expert / ACh	
	Current version=01.009	
<b>CIDType12</b>	The Type 1 and 2 Telephony Features Library	56800\CIDType12.dmo
	Status=encrypted, compressed	56800\CIDType12.drv
	Author=Processor Expert / ACh	
	Current version=01.008	

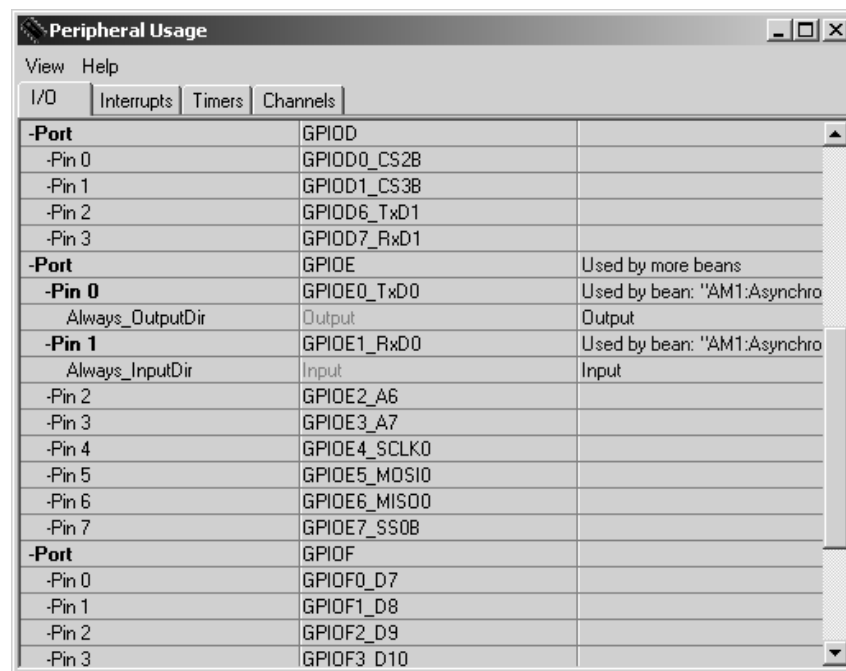
This window's View menu lets you change the display contents, such as showing driver status and information, restricting the kinds of beans the display covers, and so one.



## Peripherals Usage Inspector

The **Peripherals Usage** window (Figure 5.14) shows which bean allocates each peripheral. To open this window, select **Processor Expert > View > Peripherals Usage Inspector**, from the main-window menu bar.

Figure 5.14 Peripherals Usage Window



Peripheral Usage			
View Help			
I/O Interrupts Timers Channels			
<b>-Port</b>			
	GPIOD		
-Pin 0	GPIOD0_CS2B		
-Pin 1	GPIOD1_CS3B		
-Pin 2	GPIOD6_TxD1		
-Pin 3	GPIOD7_RxD1		
<b>-Port</b>			
	GPIOE		Used by more beans
-Pin 0	GPIOE0_TxD0		Used by bean: "AM1:Asynchro
Always_OutputDir	Output		Output
-Pin 1	GPIOE1_RxD0		Used by bean: "AM1:Asynchro
Always_InputDir	Input		Input
-Pin 2	GPIOE2_A6		
-Pin 3	GPIOE3_A7		
-Pin 4	GPIOE4_SCLK0		
-Pin 5	GPIOE5_MOSI0		
-Pin 6	GPIOE6_MISO0		
-Pin 7	GPIOE7_SS0B		
<b>-Port</b>			
	GPIOF		
-Pin 0	GPIOF0_D7		
-Pin 1	GPIOF1_D8		
-Pin 2	GPIOF2_D9		
-Pin 3	GPIOF3_D10		

The pages of this window reflect the peripheral categories: I/O, interrupts, timers, and channels. The columns of each page list peripheral pins, signal names, and the allocating beans.

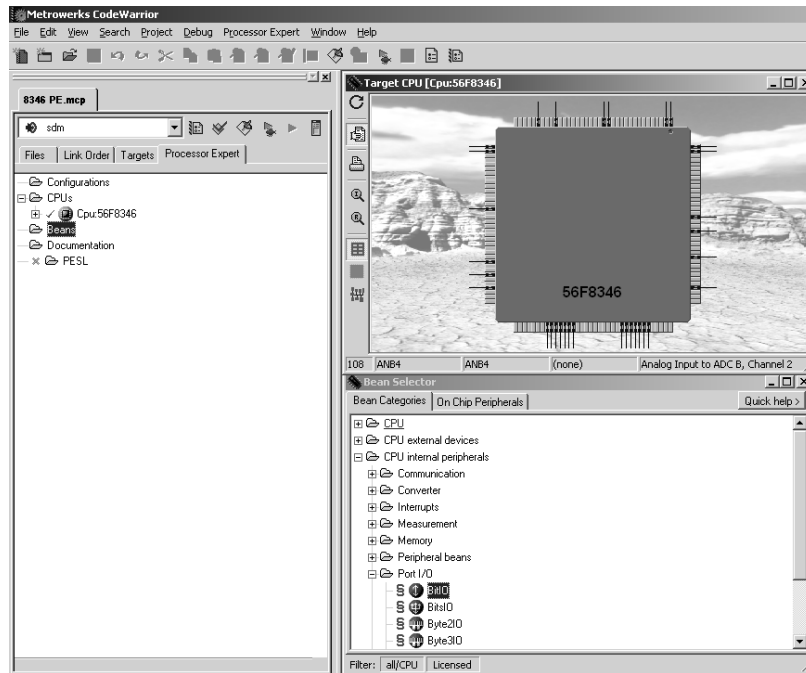
Pausing your cursor over various parts of this window brings up brief descriptions of items. This window's View menu lets you expand or collapse the display.

## Processor Expert Tutorial

This tutorial exercise generates code that flashes the LEDs of a DSP56858 development board. Follow these steps:

1. Create a project:
  - a. Start the CodeWarrior IDE, if it is not started already.
  - b. From the main-window menu bar, select **File > New**. The **New** window appears.
  - c. In the Project page, select (highlight) **Processor Expert Examples Stationery**.
  - d. In the Project name text box, enter a name for the project, such as `LEDcontrol`.
  - e. Click the **OK** button. The **New Project** window replaces the **New** window.
  - f. In the Project Stationery list, select **TestApplications > Tools > LED > 56858**.
  - g. Click the **OK** button.
  - h. Click the **OK** button. The IDE:
    - Opens the project window, docking it the left of the main window. This project window includes a Processor Expert page.
    - Opens the **Target CPU** window, as Figure 5.15 shows. This window shows the CPU package and peripherals view.
    - Opens the **Bean Selector** window, behind the **Target CPU** window.

Figure 5.15 Project, Target CPU Windows



2. Select the **sdm** external memory target.
  - a. Click the project window's Targets tab. The Targets page moves to the front of the window.
  - b. Click the target icon of the **sdm external memory** entry. The black arrow symbol moves to this icon, confirming your selection.
3. Add six BitIO beans to the project.
  - a. Click the project window's Processor Expert tab. The Processor Expert page moves to the front of the window.
  - b. Make the **Bean Selector** window visible:
    - Minimize the **Target CPU** window.
    - Select **Processor Expert > View > Bean Selector**, from the main-window menu bar.
  - c. In the Bean Categories page, expand the entry **MCU internal peripherals**.

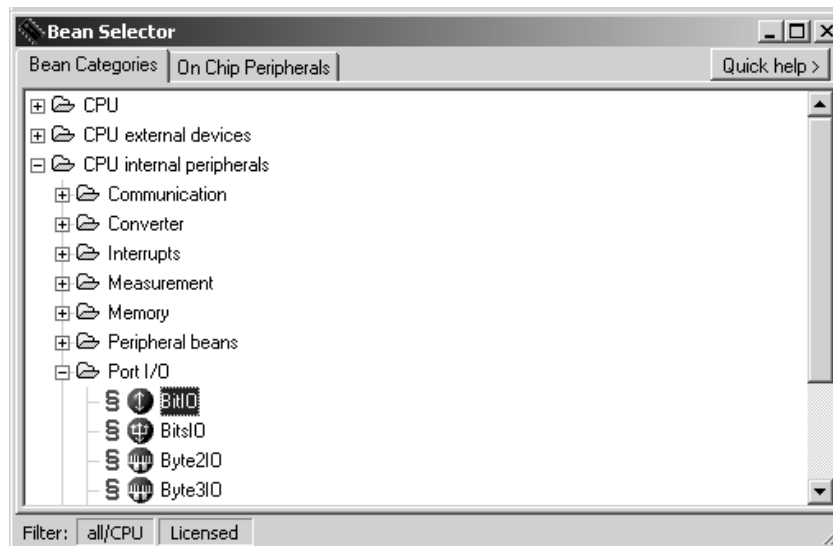
## Processor Expert Interface

### Processor Expert Tutorial

---

- d. Expand the subentry **Port I/O**.
- e. Double-click the **BitIO** bean name six times. (Figure 5.16 depicts this bean selection.) The IDE adds these beans to your project; new bean icons appear in the project window's Processor Expert page.

**Figure 5.16 Bean Selector: BitIO Selection**



---

<b>NOTE</b>	If new bean icons do not appear in the Processor Expert page, the system still may have added them to the project. Close the project, then reopen it. When you bring the Processor Expert page to the front of the project window, the page should show the new bean icons.
-------------	---

---

4. Add two ExtInt beans to the project.
  - a. In the Bean Categories page of the Bean Selector window, expand the **Interrupts** subentry.
  - b. Double-click the **ExtInt** bean name two times. The IDE adds these beans to your project; new bean icons appear in the Processor Expert page.
  - c. You may close the **Bean Inspector** window.
5. Rename the eight beans GPIO\_C0 — GPIO\_C3, GPIO\_D6, GPIO\_D7, IRQA, and IRQB.
  - a. In the project window's Processor Expert page, right-click the name of the first BitIO bean. A context menu appears.
  - b. Select **Rename Bean**. A change box appears around the bean name.

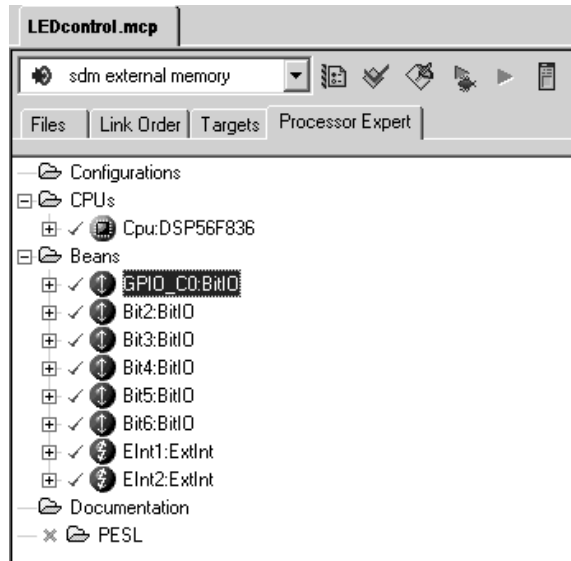
## Processor Expert Interface

Processor Expert Tutorial

---

- c. Type the new name `GPIO_C0`, then press the Enter key. The list shows the new name; as Figure 5.17 shows, this name still ends with `BitIO`.

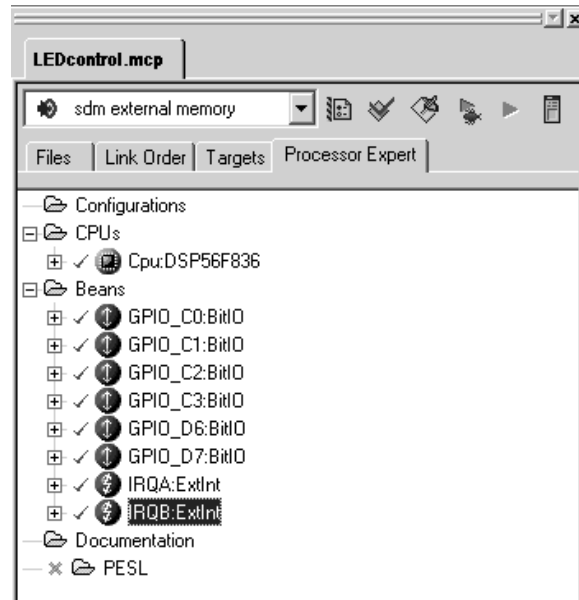
**Figure 5.17 New Bean Name**



- d. Repeat substeps a, b, and c for each of the other BitIO beans, renaming them `GPIO_C1`, `GPIO_C2`, `GPIO_C3`, `GPIO_D6`, and `GPIO_D7`.

- e. Repeat substeps a, b, and c for the two ExtInt beans, renaming them IRQA and IRQB. (Figure 5.18 shows the Processor Expert page at this point.)

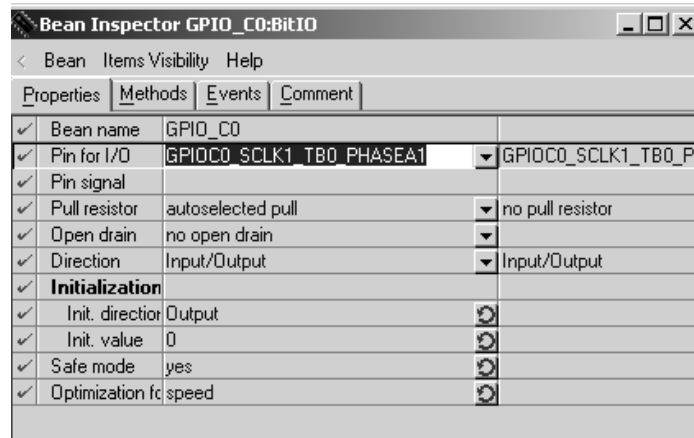
**Figure 5.18 New Bean Names**



6. Update pin associations for each bean.
  - a. In the Processor Expert page, double-click the bean name GPIO\_C0. The **Bean Inspector** window opens, displaying information for this bean.
  - b. Use standard window controls to make the middle column of the Properties page about 2 inches wide.
  - c. In the **Pin for I/O** line, click the triangle symbol of the middle-column list box. The list box opens.

- d. Use this list box to select **GPIOC0\_SCLK1\_TB0\_PHASEA1**. (Figure 5.19 depicts this selection.)

**Figure 5.19 New Pin Association**

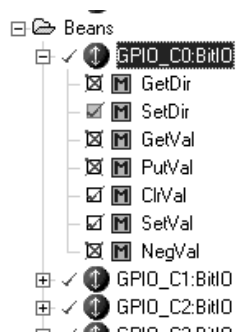


- e. In the project window's Processor Expert page, select the bean name GPIO\_C1. The Bean Inspector information changes accordingly.
- f. Use the **Pin for I/O** middle-column list box to select **GPIOC1\_MOSI1\_TB1\_PHASEB1**.
- g. Repeat substeps e and f, for bean GPIO\_C2, to change its associated pin to **GPIOC2\_MISO1\_TB2\_INDEX1**.
- h. Repeat substeps e and f, for bean GPIO\_C3, to change its associated pin to **GPIOC3\_SSA\_B\_TB3\_HOME1**.
- i. Repeat substeps e and f, for bean GPIO\_D6, to change its associated pin to **GPIOD6\_TxD1**.
- j. Repeat substeps e and f, for bean GPIO\_D7, to change its associated pin to **GPIOD7\_RxD1**.
- k. In the project window's Processor Expert page, select the bean name IRQA. The Bean Inspector information changes accordingly.
- l. Use the **Pin** middle-column list box to select **IRQA\_B**.
- m. Repeat substeps k and l, for bean IRQB, to change its associated pin to **IRQB\_B**.
- n. You may close the **Bean Inspector** window.



7. Enable BitIO SetDir, ClrVal, and SetVal functions.
  - a. In the Processor Expert page, click the plus-sign control for the GPIO\_C0 bean. The function list expands: red X symbols indicate disabled functions, green check symbols indicate enabled functions.
  - b. Double-click function symbols as necessary, so that only **SetDir**, **ClrVal**, and **SetVal** have green checks. (Figure 5.20 shows this configuration.)

**Figure 5.20 GPIO\_C3 Enabled Functions**



- c. Click the GPIO\_C0 minus-sign control. The function list collapses.
  - d. Repeat substeps a, b, and c for beans GPIO\_C1, GPIO\_C2, GPIO\_C3, GPIO\_D6, and GPIO\_D7.
8. Enable ExtInt OnInterrupt, GetVal functions.
  - a. In the Processor Expert page, click the plus-sign control for the IRQA bean. The function list expands.
  - b. Double-click function symbols as necessary, so that only **OnInterrupt** and **GetVal** have green check symbols.
  - c. Click the IRQA minus-sign control. The function list collapses.
  - d. Repeat substeps a, b, and c for bean IRQB.
9. Design (generate) project code.
  - a. From the main-window menu bar, select **Processor Expert > Code Design 'LEDcontrol.mcp.'** (This selection shows the actual name of your project.) The IDE and PEI generate several new files for your project.
  - b. You may close all windows except the project window.

10. Update file Events.c.

- a. Click the project window's Files tab. The Files page moves to the front of the window.
- b. Expand the **User Modules** folder.
- c. Double-click filename **Events.c**. An editor window opens, displaying this file's text. (Listing 5.1, at the end of this tutorial, shows this file's contents.)
- d. Find the line `IRQB_OnInterrupt()`.
- e. Above this line, enter the new line **`extern short IRQB_On;`**.
- f. Inside `IRQB_OnInterrupt()`, enter the new line **`IRQB_On ^= 1;`**.
- g. Find the line `IRQA_OnInterrupt()`.
- h. Above this line, enter the new line **`extern short IRQA_On;`**.
- i. Inside `IRQA_OnInterrupt()`, enter the new line **`IRQA_On ^= 1;`**.

---

**NOTE** Listing 5.1 shows these new lines as bold italics.

---

- j. Save and close file Events.c.

11. Update file LEDcontrol.c.

- a. In the project window's Files page, double-click filename **LEDcontrol.c** (or the actual .c filename of your project). An editor window opens, displaying this file's text.
- b. Add custom code, to utilize the beans.

---

**NOTE** Listing 5.2 shows custom entries as bold italics. Processor Expert software generated all other code of the file.

---

- c. Save and close the file.

12. Build and debug the project.

- a. From the main-window menu bar, select **Project > Make**. The IDE compiles and links your project, generating executable code.
- b. Debug your project, as you would any other CodeWarrior project.

---

This completes the Processor Expert tutorial exercise. Downloading this code to a DSP56836E development board should make the board LEDs flash in a distinctive pattern.

## Listing 5.1 File Events.c

---

```
/*
** ##### **
**   Filename   : Events.C
**
**   Project    : LEDcontrol
**
**   Processor  : DSP56F836
**
**   Beantype   : Events
**
**   Version    : Driver 01.00
**
**   Compiler   : Metrowerks DSP C Compiler
**
**   Date/Time  : 3/24/2003, 1:18 PM
**
**   Abstract   :
**
**       This is user's event module.
**       Put your event handler code here.
**
**   Settings   :
**
**   Contents   :
**
**       IRQB_OnInterrupt - void IRQB_OnInterrupt(void);
**       IRQA_OnInterrupt - void IRQA_OnInterrupt(void);
**
**
**   (c) Copyright UNIS, spol. s r.o. 1997-2002
**
**   UNIS, spol. s r.o.
**   Jundrovská 33
**   624 00 Brno
**   Czech Republic
**
**   http       : www.processorexpert.com
**   mail       : info@processorexpert.com
```

# Freescale Semiconductor, Inc.

## Processor Expert Interface Processor Expert Tutorial

---

```
**
** #####
** /
/* MODULE Events */

/*Including used modules for compilling procedure*/
#include "Cpu.h"
#include "Events.h"
#include "GPIO_C0.h"
#include "GPIO_C1.h"
#include "GPIO_C2.h"
#include "GPIO_C3.h"
#include "GPIO_D6.h"
#include "GPIO_D7.h"
#include "IRQA.h"
#include "IRQB.h"

/*Include shared modules, which are used for whole project*/
#include "PE_Types.h"
#include "PE_Error.h"
#include "PE_Const.h"
#include "IO_Map.h"

/*
** =====
**      Event      :  IRQB_OnInterrupt (module Events)
**
**      From bean   :  IRQB [ExtInt]
**      Description :
**          This event is called when the active signal edge/level
**          occurs.
**      Parameters  :  None
**      Returns     :  Nothing
** =====
** /
#pragma interrupt called
extern short IRQB_On;
void IRQB_OnInterrupt(void)
{
    IRQB_On ^=1;
    /* place your IRQB interrupt procedure body here */
}

/*
```

---

```
** =====
**      Event      :  IRQA_OnInterrupt (module Events)
**
**      From bean   :  IRQA [ExtInt]
**      Description :
**          This event is called when the active signal edge/level
**          occurs.
**      Parameters  :  None
**      Returns     :  Nothing
** =====
*/
#pragma interrupt called
extern short IRQA_On;
void IRQA_OnInterrupt(void)
{
    IRQA_On ^= 1;
    /* place your IRQA interrupt procedure body here */
}

/* END Events */

/*
** #####
**
**      This file was created by UNIS Processor Expert 03.15 for
**      the Freescale DSP56x series of microcontrollers.
**
** #####
** */
```

---

## Listing 5.2 File LEDcontrol.c

---

```
/*
** #####
**      Filename   : LEDcontrol.C
**
**      Project    : LEDcontrol
**
**      Processor  : DSP56F836
**
**      Version    : Driver 01.00
**
**      Compiler   : Metrowerks DSP C Compiler
**
**      Date/Time  : 3/24/2003, 1:18 PM
**
**      Abstract   :
**
**          Main module.
**          Here is to be placed user's code.
**
**      Settings   :
**
**
**      Contents   :
**
**          No public methods
**
**
**      (c) Copyright UNIS, spol. s r.o. 1997-2002
**
**      UNIS, spol. s r.o.
**      Jundrovská 33
**      624 00 Brno
**      Czech Republic
**
**      http       : www.processorexpert.com
**      mail       : info@processorexpert.com
**
** #####
** */
/* MODULE LEDcontrol */

/* Including used modules for compiling procedure */
#include "Cpu.h"
#include "Events.h"
```

```
#include "GPIO_C0.h"
#include "GPIO_C1.h"
#include "GPIO_C2.h"
#include "GPIO_C3.h"
#include "GPIO_D6.h"
#include "GPIO_D7.h"
#include "IRQA.h"
#include "IRQB.h"
/* Include shared modules, which are used for whole project */
#include "PE_Types.h"
#include "PE_Error.h"
#include "PE_Const.h"
#include "IO_Map.h"

/*
 * Application Description:
 *   LED program for the 56836 EVM.
 *
 *   Pattern: "Count" from 0 to 63, using LEDs to represent the bits of
the number.
 *
 *   Pressing the IRQA button flips LED order: commands that previously
went to LED1 go to LED6, and so forth.
 *   Pressing the IRQB button reverses the enabled/disabled LED states.
 */

/* global used as bitfield, to remember currently active bits, used to
 * enable/disable all LEDs. */
long    num = 0;
short   IRQA_On, IRQB_On;

/* simple loop makes LED changes visible to the eye */
void wait(int);
void wait(int count)
{
    int i;
    for (i=0; i<count; ++i);
}

/*set the given LED */
void setLED(int);
void setLED(int num)
{
    if (!IRQA_On)
```

```
{
    num = 7-num;
}
if (!IRQB_On)
{
    switch (num)
    {
        case 1: GPIO_C0_ClrVal(); break;
        case 2: GPIO_C1_ClrVal(); break;
        case 3: GPIO_C2_ClrVal(); break;
        case 4: GPIO_C3_ClrVal(); break;
        case 5: GPIO_D6_ClrVal(); break;
        case 6: GPIO_D7_ClrVal(); break;
    }
}
else
{
    switch (num)
    {
        case 1: GPIO_C0_SetVal(); break;
        case 2: GPIO_C1_SetVal(); break;
        case 3: GPIO_C2_SetVal(); break;
        case 4: GPIO_C3_SetVal(); break;
        case 5: GPIO_D6_SetVal(); break;
        case 6: GPIO_D7_SetVal(); break;
    }
}
}

/* clear the given LED */
void clrLED(int);
void clrLED(int num)
{
    if (!IRQA_On)
    {
        num = 7-num;
    }
    if (IRQB_On)
    {
        switch (num)
        {
            case 1: GPIO_C0_ClrVal(); break;
            case 2: GPIO_C1_ClrVal(); break;
            case 3: GPIO_C2_ClrVal(); break;
            case 4: GPIO_C3_ClrVal(); break;
        }
    }
}
```



---

```

        case 5: GPIO_D6_ClrVal(); break;
        case 6: GPIO_D7_ClrVal(); break;
    }
}
else
{
    switch (num)
    {
        case 1: GPIO_C0_SetVal(); break;
        case 2: GPIO_C1_SetVal(); break;
        case 3: GPIO_C2_SetVal(); break;
        case 4: GPIO_C3_SetVal(); break;
        case 5: GPIO_D6_SetVal(); break;
        case 6: GPIO_D7_SetVal(); break;
    }
}
}

#define CLEARLEDS    showNumberWithLEDS(0)
/* method to set each LED status to reflect the given number/bitfield */
void shwNumberWithLEDS(long);
void showNumberWithLEDS(long num)
{
    int i;
    for (i=0; i<6; ++i)
    {
        if ((num>>i) & 1
            setLED(i+1);
        else
            clrLED(i+1);
    }
}

/* Pattern: "Count" from 0 to 63 in binary using LEDs to represent bits
of the current number. 1 = enabled LED, 0 = disabled LED. */
void pattern();
void pattern()
{
    long i;
    int j;

    for (i=0; i<=0b1111111; ++i)
    {
        showNumberWithLEDS(i);
        wait(100000);
    }
}

```

# Freescal Semiconductor, Inc.

## Processor Expert Interface

### Processor Expert Tutorial

---

```
    }
}

void main(void)
{
    /** Processor Expert internal initialization. DON'T REMOVE THIS
CODE!!! ***/
    PE_low_level_init();
    /** End of Processor Expert internal initialization.          ***/

    /*Write your code here*/
#pragma warn_possunwant off

    IRQA_On = IRQA_GetVal() ? 1 : 0;
    IRQB_On = IRQB_GetVal() ? 1 : 0;

    for(;;) {

        CLEARLEDS;
        pattern();

    }

#pragma warn_possunwant reset
}

/* END LEDcontrol */
/*
** #####
** This file was created by UNIS Processor Expert 03.15 for
** the Freescale DSP56x series of microcontrollers.
** #####
**/
```

---

# C for DSP56800E

---

This chapter explains considerations for using C with the DSP56800E processor. Note that the DSP56800E processor does not support:

- The C++ language
- Standard C trigonometric and algebraic floating-point functions (such as sine, cosine, tangent, and square root)

Furthermore, C pointers allow access only to X memory.

---

<b>NOTE</b>	The DSP56800E MSL implements a few trigonometric and algebraic functions, but these are mere examples that the DSP56800E does not support.
-------------	--

---

This chapter contains these sections:

- Number Formats
- Calling Conventions and Stack Frames
- User Stack Allocation
- Data Alignment Requirements
- Variables in Program Memory
- Packed Structures Support
- Code and Data Storage
- Large Data Model Support
- Optimizing Code
- Deadstripping and Link Order
- Working with Peripheral Module Registers
- Generating MAC Instruction Set

## Number Formats

This section explains how the CodeWarrior compiler implements ordinal and floating-point number types for 56800E processors. For more information, read `limits.h` and `float.h`, under the M56800E Support folder.

Table 6.1 shows the sizes and ranges of ordinal data types.

**Table 6.1 56800E Ordinal Types**

Type	Option Setting	Size (bits)	Range
char	Use Unsigned Chars is disabled in the C/C++ Language (C Only) settings panel	8	-128 to 127
	Use Unsigned Chars is enabled	8	0 to 255
signed char	n/a	8	-128 to 127
unsigned char	n/a	8	0 to 255
short	n/a	16	-32,768 to 32,767
unsigned short	n/a	16	0 to 65,535
int	n/a	16	-32,768 to 32,767
unsigned int	n/a	16	0 to 65,535
long	n/a	32	-2,147,483,648 to 2,147,483,647
unsigned long	n/a	32	0 to 4,294,967,295
pointer	small data model ("Large Data Model" is disabled in the M56800E Processor settings panel)	16	0 to 65,535
	large data model ("Large Data Model" is enabled)	24	0 to 16,777,215

Table 6.2 shows the sizes and ranges of the floating-point types.

**Table 6.2 M56800E Floating-Point Types**

Type	Size (bits)	Range
float	32	1.17549e-38 to 3.40282e+38
short double	32	1.17549e-38 to 3.40282e+38
double	32	1.17549e-38 to 3.40282e+38
long double	32	1.17549e-38 to 3.40282e+38

## Calling Conventions and Stack Frames

The DSP56800E compiler stores data and call functions differently than the DSP56800 compiler does. Advantages of the DSP56800E method include: more registers for parameters and more efficient byte storage.

### Passing Values to Functions

The compiler uses registers A, B, R1, R2, R3, R4, Y0, and Y1 to pass parameter values to functions. Upon a function call, the compiler scans the parameter list from left to right, using registers for these values:

- The first two 16-bit integer values — Y0 and Y1.
- The first two 32-bit integer or float values — A and B.
- The first four pointer parameter values — R2, R3, R4, and R1 (in that order).
- The third and fourth 16-bit integer values — A and B (provided that the compiler does not use these registers for 32-bit parameter values).
- The third 16-bit integer value — B (provided that the compiler does not use this register for a 32-bit parameter value).

The compiler passes the remaining parameter values on the stack. The system increments the stack by the total amount of space required for memory parameters. This incrementing must be an even number of words, as the stack pointer (SP) must be continuously long-aligned. The system moves parameter values to the stack from left to right, beginning with the stack location closest to the SP. Because a long parameter must begin at an even address, the compiler introduces one-word gaps before long parameter values, as appropriate.

## Returning Values From Functions

The compiler returns function results in registers A, R0, R2, and Y0:

- 16-bit integer values — Y0.
- 32-bit integer or float values — A.
- All pointer values — R2.
- Structure results — R0 contains a pointer to a temporary space allocated by the caller. (The pointer is a hidden parameter value.)

Additionally, the compiler:

- Reserves R5 for the stack frame pointer when a function makes a dynamic allocation. (This is the original stack pointer before allocations.) Otherwise, the compiler saves R5 across function calls.
- Saves registers C10 and D10 across function calls.
- Does not save registers C2 and D2 across function calls.

## Volatile and Non-Volatile Registers

Values in *non-volatile* registers can be saved across functions calls. Another term for such registers is *saved over a call* registers (SOCs).

Values in *volatile* registers cannot be saved across functions calls. Another term for such registers is *non-SOC* registers.

Table 6.3 lists both the volatile and non-volatile registers.

**Table 6.3 Volatile and Non-Volatile Registers**

Unit	Register	Size	Type	Comments
Arithmetic Logic Unit (ALU)	Y1	16	Volatile (non-SOC)	
	Y0	16	Volatile (non-SOC)	
	Y	32	Volatile (non-SOC)	
	X0	16	Volatile (non-SOC)	
	A2	4	Volatile (non-SOC)	
	A1	16	Volatile (non-SOC)	
	A0	16	Volatile (non-SOC)	

**Table 6.3 Volatile and Non-Volatile Registers (*continued*)**

Unit	Register	Size	Type	Comments
Arithmetic Logic Unit (ALU) (continued)	A10	32	Volatile (non-SOC)	
	A	36	Volatile (non-SOC)	
	B2	4	Volatile (non-SOC)	
	B1	16	Volatile (non-SOC)	
	B0	16	Volatile (non-SOC)	
	B10	32	Volatile (non-SOC)	
	B	36	Volatile (non-SOC)	
	C2	4	Volatile (non-SOC)	
	C1	16	Non-Volatile (SOC)	
	C0	16	Non-Volatile (SOC)	
	C10	32	Non-Volatile (SOC)	
	C	36	Volatile (non-SOC)	Includes volatile register C2.
	D2	4	Volatile (non-SOC)	
	D1	16	Non-Volatile (SOC)	
	D0	16	Non-Volatile (SOC)	
	D10	32	Non-Volatile (SOC)	
	D	36	Volatile (non-SOC)	Includes volatile register D2.
Address Generation Unit (AGU)	R0	24	Volatile (non-SOC)	
	R1	24	Volatile (non-SOC)	
	R2	24	Volatile (non-SOC)	
	R3	24	Volatile (non-SOC)	
	R4	24	Volatile (non-SOC)	
	R5	24	Non-volatile (SOC)	If the compiler uses R5 as a pointer, it becomes a volatile register — its value can be saved over called functions.
	N	24	Volatile (non-SOC)	

# Freescale Semiconductor, Inc.

## C for DSP56800E

### Calling Conventions and Stack Frames

**Table 6.3 Volatile and Non-Volatile Registers (*continued*)**

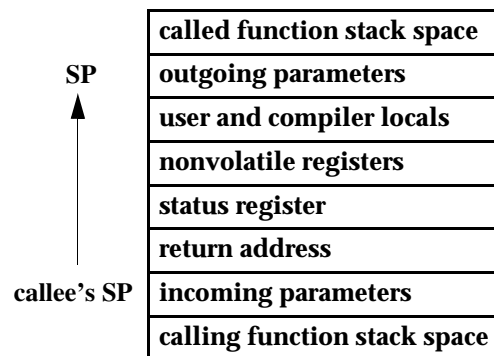
Unit	Register	Size	Type	Comments
Address Generation Unit (AGU) (continued)	SP	24	Volatile (non-SOC)	
	N3	16	Volatile (non-SOC)	
	M01	16	Volatile (non-SOC)	Certain registers must keep specific values for proper C execution — set this register to 0xFFFF.
Program Controller	PC	21	Volatile (non-SOC)	
	LA	24	Volatile (non-SOC)	
	LA2	24	Volatile (non-SOC)	
	HWS	24	Volatile (non-SOC)	
	FIRA	21	Volatile (non-SOC)	
	FISR	13	Volatile (non-SOC)	
	OMR	16	Volatile (non-SOC)	Certain registers must keep specific values for proper C execution — in this register, set the CM bit.
	SR	16	Volatile (non-SOC)	
	LC	16	Volatile (non-SOC)	
	LC2	16	Volatile (non-SOC)	



## Stack Frame and Alignment

Figure 6.1. depicts generation of the stack frame. The stack grows upward, so pushing data onto the stack increments the stack pointer's address value.

**Figure 6.1 Stack Frame**



The stack pointer (SP) is a 24-bit register, always treated as a word pointer. During a function execution, the stable position for the SP is at the top of the user and compiler locals. The SP increases during the call if the stack is used for passed parameters.

The software stack supports structured programming techniques, such as parameter passing to subroutines and local variables. These techniques are available for both assembly-language and high-level-language programming. It is possible to support passed parameters and local variables for a subroutine at the same time within the stack frame.

The compiler stores local data by size. It stores smaller data closest to the SP, exploiting SP addressing modes that have small offsets. This means that the compiler packs all bytes two per word near the stack pointer. It packs the block of words next, then blocks of longs. Aggregates (structs and arrays) are farthest from the stack pointer, not sorted by size.

---

**NOTE** When a function makes a dynamic allocation, the compiler reserves R5 as a stack frame pointer. (This is the stack pointer before allocations.)

---

The compiler always must operate with the stack pointer long aligned. This means that:

- The start-up code in the runtime first initializes the stack pointer to an odd value.
- At all times after that, the stack pointer must point to an odd word address.
- The compiler never generates an instruction that adds or subtracts an odd value from the stack pointer.
- The compiler never generates a MOVE.W or MOVEU.W instruction that uses the X:(SP)+ or X:(SP)- addressing mode.

## User Stack Allocation

The 56800E compilers build frames for hierarchies of function calls using the stack pointer register (SP) to locate the next available free X memory location in which to locate a function call's frame information. There is usually no explicit frame pointer register. Normally, the size of a frame is fixed at compile time. The total amount of stack space required for incoming arguments, local variables, function return information, register save locations (including those in pragma interrupt functions) is calculated and the stack frame is allocated at the beginning of a function call.

Sometimes, you may need to modify the SP at runtime to allocate temporary local storage using inline assembly calls. This invalidates all the stack frame offsets from the SP used to access local variables, arguments on the stack, etc. With the User Stack Allocation feature, you can use inline assembly instructions (with some restrictions) to modify the SP while maintaining accurate local variable, compiler temps, and argument offsets, i.e., these variables can still be accessed since the compiler knows you have modified the stack pointer.

The User Stack Allocation feature is enabled with the `#pragma check_inline_sp_effects [on|off|reset]` pragma setting. The pragma may be set on individual functions. By default the pragma is off at the beginning of compilation of each file in a project.

The User Stack Allocation feature allows you to simply add inline assembly modification of the SP anywhere in the function. The restrictions are straight-forward:

1. The SP must be modified by the same amount on all paths leading to a control flow merge point
2. The SP must be modified by a literal constant amount. That is, address modes such as "(SP)+N" and direct writes to SP are not handled.
3. The SP must remain properly aligned.

4. You must not overwrite the compiler's stack allocation by decreasing the SP into the compiler allocated stack space.

Point 1 above is required when you think about an if-then-else type statement. If one branch of a decision point modifies the SP one way and the other branch modifies SP another way, then the value of the SP is run-time dependent, and the compiler is unable to determine where stack-based variables are located at run-time. To prevent this from happening, the User Stack Allocation feature traverses the control flow graph, recording the inline assembly SP modifications through all program paths. It then checks all control flow merge points to make sure that the SP has been modified consistently in each branch converging on the merge point. If not, a warning is emitted citing the inconsistency.

Once the compiler determined that inline SP modifications are consistent in the control flow graph, the SP's offsets used to reference local variables, function arguments, or temps are fixed up with knowledge of inline assembly modifications of the SP. Note, you may freely allocate local stack storage:

1. As long as it is equally modified along all branches leading to a control flow merge point.
2. The SP is properly aligned. The SP must be modified by an amount the compiler can determine at compile time.

A single new pragma is defined. `#pragma check_inline_sp_effects [on|off|reset]` will generate a warning if the user specifies an inline assembly instruction which modifies the SP by a run-time dependent amount. If the pragma is not specified, then stack offsets used to access stack-based variables will be incorrect. It is the user's responsibility to enable `#pragma check_inline_sp_effects`, if they desire to modify the SP with inline assembly and access local stack-based variables. Note this pragma has no effect in function level assembly functions or separate assembly only source files (.asm files).

In general, inline assembly may be used to create arbitrary flow graphs and not all can be detected by the compiler.

For example:

---

```
REP #3
ADDA #2,SP
```

---

This example would modify the SP by three, but the compiler would only see a modification of one. Other cases such as these might be created by the user using inline jumps or branches. These are dangerous constructs and are not detected by the compiler.

In cases where the SP is modified by a run-time dependent amount, a warning is issued.

## **Listing 6.1 Example 1 – Legal modification of SP Using Inline Assembly**

---

```
#define EnterCritical() { asm(adda #2,SP);\n                        asm(move.l  SR,X:(SP)+);\n                        asm(bfset #0x0300,SR);\n                        asm(nop);\n                        asm(nop);}\n\n#define ExitCritical() { asm(deca.l SP);\n                        asm(move.l x:(SP)-,SR);\n                        asm(nop);\n                        asm(nop);}\n\n#pragma check_inline_sp_effects on\n\nint func()\n{\n    int a=1, b=1, c;\n\n    EnterCritical();\n\n    C = a+b;\n\n    ExitCritical();\n\n}
```

---

This case will work because there are no control flow merge points. SP is modified consistently along all paths from the beginning to the end of the function and is properly aligned.

## **Listing 6.2 Example 2 – Illegal Modification of SP using Inline Assembly**

---

```
#define EnterCritical() { asm(adda #2,SP);\n                        asm(move.l  SR,X:(SP)+);\n                        asm(bfset #0x0300,SR);\n                        asm(nop);\n                        asm(nop);}\n\n#define ExitCritical() { asm(deca.l SP);\n                        asm(move.l x:(SP)-,SR);\n
```

---

---

```

        asm(nop); \
        asm(nop); }

#pragma check_inline_sp_effects on

int func()
{
    int a=1, b=1, c;

    if (a)
    {
        EnterCritical();

        c = a+b;

    }
    else {
        c = b++;
    }

    ExitCritical();

    return (b+c);
}

```

---

This example will generate the following warning because the SP entering the 'ExitCritical' macro is different depending on which branch is taken in the if. Therefore, accesses to variables a, b, or c may not be correct.

---

```

Warning : Inconsistent inline assembly modification of SP in this
function.
M56800E_main.c line 29      ExitCritical();

```

---

## Listing 6.3 Example 3 – Modification of SP by a Run-time Dependent Amount

---

```

#define EnterCritical() { asm(adda R0,SP); \
                        asm(move.l  SR,X:(SP)+); \
                        asm(bfset  #0x0300,SR); \
                        asm(nop); \
                        asm(nop); }

#define ExitCritical() { asm(deca.l SP); \
                        asm(move.l  X:(SP)-,SR); \
                        asm(nop); \

```

---

## C for DSP56800E User Stack Allocation

---

```
asm(nop); }

#pragma check_inline_sp_effects on
int func()
{
    int a=1, b=1, c;

    if (a)
    {
        EnterCritical();

        c = a+b;

    }
    else {
        EnterCritical();
        c = b++;
    }

    return (b+c);
}
```

---

This example will generate the following warning:

---

```
Warning : Cannot determine SP modification value at compile time
M56800E_main.c line 20      EnterCritical();
```

---

This example is not legal since the SP is modified by run-time dependent amount.

If all inline assembly modifications to the SP along all branches are equal approaching the exit of a function, it is not necessary to explicitly deallocate the increased stack space. The compiler “cleans up” the extra inline assembly stack allocation automatically at the end of the function.

### Listing 6.4 Example 4 – Automatic Deallocation of Inline Assembly Stack Allocation

---

```
#pragma check_inline_sp_effects on
int func()
{
    int a=1, b=1, c;

    if (a)
```

---

```
    {  
        EnterCritical();  
  
        c = a+b;  
    }  
    else {  
        EnterCritical();  
        c = b++;  
    }  
  
    return (b+c);  
}
```

---

This example does not need to call the 'ExitCritical' macro because the compiler will automatically clean up the extra inline assembly stack allocation.

## Data Alignment Requirements

The data alignment rules for DSP56800E stack and global memory are:

- Bytes — byte boundaries.  
Exception: bytes passed on the stack are always word-aligned, residing in the lower bytes.
- Words — word boundaries.
- Longs, floats, and doubles — double-word boundaries:
  - Least significant word is always on an even word address.
  - Most significant word is always on an odd word address.
  - Long accesses through pointers in AGU registers (for example, R0 through R5 or N) point to the least significant word. That is, the address is even.
  - Long accesses through pointers using SP point to the most significant word. That is, the address in SP is odd.
- Structures — word boundaries (not byte boundaries).

---

<b>NOTE</b>	A structure containing only bytes still is word aligned.
-------------	--

---

- Structures — double-word boundaries if they contain 32-bit elements, or if an inner structure itself is double-word aligned.
- Arrays — the size of one array element.

## Word and Byte Pointers

The alignment requirements explained above determine how the compiler uses DSP56800E byte and word pointers to implement C pointer types. The compiler uses:

- Word pointers for all structures
- The SP to access the stack resident data of all types:
  - Bytes
  - Shorts
  - Longs
  - Floats
  - Doubles
  - Any pointer variables
- Word pointers to access:
  - Shorts
  - Longs
  - Any pointer variables
- Byte pointers for:
  - Single global or static byte variable, if accessed through a pointer using `X:(Rn)`
  - Global or static array of byte variables

The compiler does not use pointers to access scalar global or static byte variables directly by their addresses. Instead, it uses an instruction with a `.BP` suffix:

```
MOVE [U] .BP    X:xxxx, <dest>
MOVE .BP        <src>, X:xxxx
```

## Reordering Data for Optimal Usage

The compiler changes data order, for optimal usage. The data reordering follows these guidelines:



- Reordering is mandatory if local variables are allocated on the stack.
- The compiler does not reorder data for parameter values passed in memory (instead of being passed in registers).
- The compiler does not reorder data when locating fields within a structure.

## Variables in Program Memory

This feature allows the programmer full flexibility in deciding the placement of variables in memory. Variables can be now declared as part of the program memory, using a very simple and intuitive syntax. For example:

```
__pmem int c; // 'c' is an integer that will be stored in program memory.
```

This feature is very useful when data memory is tight, because some or all of the data can be moved to program memory. It can be handled exactly the same way as normal data. This is almost completely transparent to the programmer, with a few exceptions that will be presented in the next paragraphs.

The CPU architecture only allows post increment addressing of words (16 bit data) in program memory. While the compiler circumvents this restriction and allows full access to all data types in program memory, the performance is decreased. If placement of some variables in program memory is needed, and at the same time the execution speed is important, here are some pointers that can be used to organize the code:

- Try to keep all variables that are used in a loop (the loop counter included) in data memory. This condition becomes more important as the loop nesting level increases.
- If possible, place only int (16-bit) data in program memory. Data types with different dimensions are accessed via sequences of code rather than single instructions. 16-bit data is fastest, followed by 32-bit data and 8-bit data.
- Data in program memory can be loaded and stored in a limited number of DALU registers. Because of this, a number of register save/restore sequences can appear if there are not enough available DALU registers. This could be a problem with computational intensive code, because the operations does not take place only in registers anymore and the code will be slower. This can be avoided by using as many variables in data memory as possible.

## Declaring Program Memory Variables

A program memory variable is declared using the `__pmem` qualifier. Here are some examples:

## C for DSP56800E

### Variables in Program Memory

---

```
typedef struct // simple structure declaration
{
    int i;
    char *p;
    long l;
} test;

__pmem int ip1 = 5; // initialized int in program memory
__pmem int ip2; // uninitialized int in program memory
int *__pmem ppx1; // pointer in program memory to int in data memory
__pmem int *__pmem ppp1; // pointer in program memory to int in program
                        //memory
__pmem int parr[ 100 ]; // array in program memory
__pmem tests sp; // structure in program memory
__pmem int aap[ 2 ][ 2 ]; // two dimensional array in program memory
__pmem int *pxp1; // pointer in data memory to int in program memory
```

---

## Using Variables in Program Memory

Variables in program memory can be used almost exactly like variables in data memory. The exceptions are presented below:

- the `__pmem` qualifier can't be used in a structure declaration, because a structure can have all its members either in program memory or in data memory, but not in both memory spaces. The compiler will issue an error message in this case. For example:

```
typedef struct // simple structure declaration
{
    int i;
    char __pmem *p; // error, __pmem not allowed here
    long l;
} test;
```

---

- the compiler will signal an error when an implicit conversion between a pointer to data in data memory and a pointer to data in program memory is attempted. For example, using the previous definitions, the compiler gives an error for this assignment:

---

```
pxp1 = ppx1;
```

---

Explicit conversions are allowed, but they should be used with care. An explicit conversion for the previous assignment that is accepted by the compiler is given below:

---

```
pxp1 = ( __pmem int * )ppx1;
```

---

Another consequence of this restriction is that an important part of the MSL functions that have at least an argument that is a pointer will not work with variables in program memory. For example:

---

```
char *c1; // pointer in data memory to char in data memory
char __pmem *c2; // pointer in data memory to char in program memory
strcat( c1, c2 ); // error, the second argument can't be
                  //converted to 'const char *'
```

---

If variable argument lists are used, this problem is generally hidden. The program is compiled with no errors from the compiler, but it doesn't work as expected. The most common example is the `printf` function:

---

```
char *c1 = "xmem";           // pointer in data memory to char in data
                              //memory
char __pmem *c2 = "pmem";    // pointer in data memory to char in program
                              //memory

printf( "%s\n", c1 );        // works as expected
printf( "%s\n", c2 );        // doesn't work as expected
```

---

Here, the type of the arguments is lost because `printf` uses a variable argument list. Thus the compiler can not signal a type mismatch and the program will compile without errors, but it won't work as expected, because `printf` assumes that all the data is stored in data memory.

---

## Linking with Variables in Program Memory

The compiler creates special sections in the output file for variables in program memory. This is a description of all data in program memory sections:

- .data.pmem (initialized program memory data)
- .const.data.pmem (constant program memory data)
- .bss.pmem (uninitialized program memory data).

The following sections are also generated if you choose to generate separate sections for char data:

- .data.char.pmem (initialized program memory chars)
- .const.data.char.pmem (constant program memory chars)
- .bss.char.pmem (uninitialized program memory chars)

These sections are used in the linker command file just like normal sections. A typical linker command file for a program that uses data in program memory looks like this:

```
MEMORY
{
    .p_RAM          (RWX) : ORIGIN = 0x0082,   LENGTH = 0xFF3E
    .p_reserved_regs (RWX) : ORIGIN = 0xFFC0,   LENGTH = 0x003F
    .p_RAM2          (RWX) : ORIGIN = 0xFFFF,   LENGTH = 0x0000
    .x_RAM          (RW)  : ORIGIN = 0x0001,   LENGTH = 0x7FFE
                                # SDM xRAM limit is 0x7FFF
}

SECTIONS
{
    .application_code :
    {
        # .text sections

        * (.text)
        * (rtlib.text)
        * (fp_engine.text)
        * (user.text)
        * (.data.pmem)          # program memory initialized data
        * (.const.data.pmem)    # program memory constant data
        * (.bss.pmem)           # program memory uninitialized data
    } > .p_RAM

    .data :
    {
        # .data sections
```

---

```

* (.const.data.char) # used if "Emit Separate Char Data
                      # Section" enabled
* (.const.data)
* (fp_state.data)
* (rtlib.data)
* (.data.char)       # used if "Emit Separate Char Data
                      # Section" enabled
* (.data)

# .bss sections

* (rtlib.bss.lo)
* (rtlib.bss)
. = ALIGN(1);
_START_BSS = .;
* (.bss.char)        # used if "Emit Separate Char Data
                      # Section" enabled
* (.bss)
_END_BSS = .;

# setup the heap address

. = ALIGN(4);
_HEAP_ADDR = .;
_HEAP_SIZE = 0x100;
_HEAP_END = _HEAP_ADDR + _HEAP_SIZE;
. = _HEAP_END;

# setup the stack address

_min_stack_size = 0x200;
_stack_addr = _HEAP_END;
_stack_end = _stack_addr + _min_stack_size;
. = _stack_end;

# export heap and stack runtime to libraries

F_heap_addr = _HEAP_ADDR;
F_heap_end = _HEAP_END;
F_lstack_addr = _HEAP_END;
F_start_bss = _START_BSS;
F_end_bss = _END_BSS;
} > .x_RAM
}

```

---

## Packed Structures Support

This feature allows a better data layout for long data type members in structures. It is actually a particular case for the feature that allows all long data types to be aligned only on 2-bytes of boundary. The default alignment for this case is overridden so that their alignment is set to a word (2 bytes) boundary instead of a long word (4 bytes) boundary. The reason behind the default alignment rule is that accessing a long data type at addresses that are not aligned on a 4-byte boundary results in an unaligned access exception.

The DSP56800E compiler only supports the following syntax:

```
__attribute__((packed, aligned(2)))
```

which allows data types that are 4-byte long to be aligned on a 2-byte boundary.

This is the case for long data type and pointers to any data type in the large-memory-model.

Consider the following two data structures, with identical data members, one packed and the other unpacked, and assume compilation of the large data model:

---

```
struct __attribute__((packed, aligned(2))) STag1 {  
    char c;  
    long l;  
    char c1;  
    char* pc;  
}S1;  
  
struct STag2 {  
    char c;  
    long l;  
    char c1;  
    char* pc;  
}S2;
```

---

The size of S1 is 12 bytes, while the size of S2 is 16 bytes.

---

<b>NOTE</b>	It is your responsibility to inform the compiler about pointer access to misaligned structure members.
-------------	--

---

For instance one should declare the following:

---

```

long __attribute__((packed, aligned(2))) lg; //packed long
char* __attribute__((packed, aligned(2))) in; // packed pointer
//to char
char* __attribute__((packed, aligned(2))) * pin; // pointer to
//packed pointer to char
char* __attribute__((packed, aligned(2)))
    * __attribute__((packed, aligned(2))) ppin;
// packed pointer to packed pointer to char

struct __attribute__((packed, aligned(2))) STag {
    // __attribute__ propagated to member level
    char c; // offset 0
    long l; // offset 2
    char cl; // offset 6
    char* pc; /* assume LMM*/ // offset 8
};

```

---

The compiler will issue an error if packed data is accessed by regular pointers. For example:

---

```

long __attribute__((packed, aligned(2))) l1;
long * pl = &l1; // results in error : illegal implicit
//conversion

```

---

## Code and Data Storage

The DSP56800E processor has a dual Harvard architecture with separate CODE (P: memory) and DATA (X: memory) memory spaces. Table 6.4. shows the sizes and ranges of these spaces, as well as the range of character data within X memory, for both the small and large memory models. (You may need to use the **ELF Linker and Command Language** or **M56800E Linker** settings panel to specify how the project-defined sections map to real memory.)

**Table 6.4 Code and Data Memory Ranges**

Section	Small Model		Large Model	
	Size	Range (Word Address)	Size	Range (Word Address)
CODE (P: memory)	128 KB	0 - 0xFFFF	1 MB	0 - 0x7FFFF
DATA (X: memory)	128 KB	0 - 0xFFFF	32 MB	0 - 0xFFFFF
DATA (X: memory) character data	64 KB	0 - 0x7FFF	16 MB	0 - 0x7FFFF

A peculiarity of the DSP56800E architecture is byte addresses for character (1-byte) data, but word addresses for data of all other types. To calculate a byte address, multiply the word address by 2. An address cannot exceed the maximum physical address, so placing character data in the upper half of memory makes the data unaddressable. (Address registers have a fixed width.)

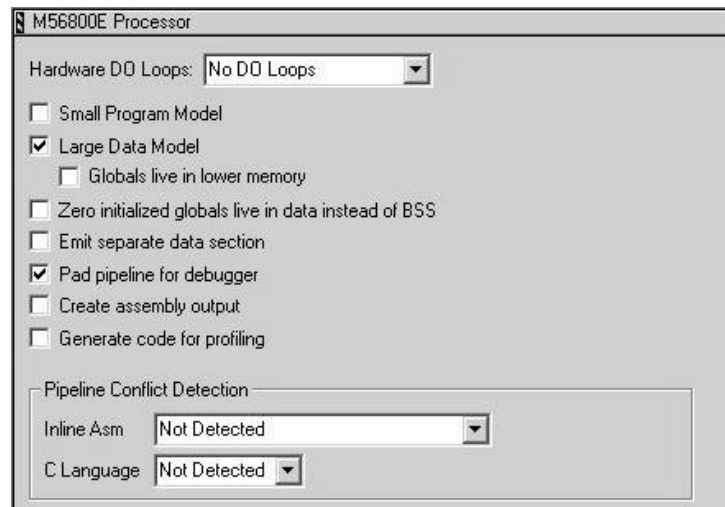
For example, in the small memory model (maximum data address: 64 KB), placing character data at 0x8001 requires an access address of 0x10002. But this access address does not fit into 16-bit storage, as the small data memory model requires. Under your control, the compiler increases flexibility by placing all character data into specially-named sections as described in “Emit separate character data section checkbox.”. You can locate these sections in the lower half of the memory map, making sure that the data can be addressed.



## Large Data Model Support

The DSP56800E extends the DSP56800 data addressing range, by providing 24-bit address capability to some instructions. 24-bit address modes allow user accesses beyond the 64K-word boundary of 16-bit addressing. To control large data memory model support, use the M56800E Processor panel (Figure 6.2). See “M56800E Processor” for explanations of this panel’s elements.

**Figure 6.2 M56800E Processor Panel: Large Data Model**



*Extended data* is data located beyond the 16-bit address boundary — as if it exists in extended (upper) memory. Memory located below the 64K boundary is *lower memory*.

The compiler default arrangement is using 16-bit addresses for all data accesses. This means that absolute addresses (X:xxxx addressing mode) are limited to 16 bits. Direct addressing or pointer registers load or store 16-bit addresses. Indexed addressing indexes are 16-bit quantities. The compiler treats data pointers as 16-bit pointers that you may store in single words of memory.

If the large data memory model is enabled, the compiler accesses all data by 24-bit addressing modes. It treats data pointers as 24-bit quantities that you may store in two words of memory. Absolute addressing occurs as 24-bit absolute addresses. Thus, you may access the entire 24-bit data memory, locating data objects anywhere.

You do not need to change C source code to take advantage of the large data memory model.

Examples in DSP56800E assembly code of extended data addressing are:

## Extended Data Addressing Example

Consider the code of Listing 6.5:

### Listing 6.5 Addressing Extended Data

---

```
move.w x:0x123456,A1      ; move int using 24 bit absolute address
tst.l  x:(R0-0x123456)    ; test a global long for zero using 24-bit
                           ; pointer indexed addressing
move.l r0,x:(R0)+         ; r0 stored as 24-bit quantity
cmpa  r0,r1               ; compare pointer registers as 24 bit
                           ; quantities
```

---

The large data memory model is convenient because you can place data objects anywhere in the 24-bit data memory map. But the model is inefficient because extended data addressing requires more program memory and additional execution cycles.

However, all global and static data of many target applications easily fits within the 64 KB word memory boundary. With this in mind, you can check the **Globals live in lower memory** checkbox of the M56800E Processor settings panel. This tells the compiler to access global and static data with 16-bit addresses, but to use 24-bit addressing for all pointer and stack operations. This arrangement combines the flexibility of the large data memory model with the efficiency of the small data model's access to globals and statics.

---

<b>NOTE</b>	If you check the Globals live in lower memory checkbox, be sure to store data in lower memory.
-------------	--

---

## Accessing Data Objects Examples

Table 6.5 and Table 6.6 show appropriate ways to access a global integer and a global pointer variable. The first two columns of each table list states of two checkboxes, **Large Data Model** and **Globals live in lower memory**. Both checkboxes are in the M56800E Processor settings panel. Note that the first enables the second.

Table 6.5 lists ways to access a global integer stored at address X:0x1234.

```
int gl;
```

**Table 6.5 Accessing a Global Integer**

Large Data Model checkbox	Globals live in lower memory checkbox	Instruction	Comments
Clear	Clear	move.w X:0x1234,y0	Default values
Checked	Clear	move.w X:0x001234,y0	
Clear	Checked	Combination not allowed	
Checked	Checked	move.w X:0x1234,y0	Global accesses use 16-bit addressing

Table 6.6 lists ways to load a global pointer variable, at X:0x4567, into an address register.

```
int * gp1;
```

**Table 6.6 Loading a Global Pointer Variable**

Large Data Model checkbox	Globals live in lower memory checkbox	Instruction	Comments
Clear	Clear	move.w X:0x4567,r0	Default 16-bit addressing, 16-bit pointer value
Checked	Clear	move.l X:0x004567,r0	24-bit addressing, pointer value is 24-bit
Clear	Checked	Combination not allowed	
Checked	Checked	move.l X:0x4567,r0	16-bit addressing, pointer value is 24-bit

## External Library Compatibility

If you enable the large data model when the compiler builds your main application, external libraries written in C also must be built with the large data model enabled. The linker enforces this requirement, catching global objects located out of range for particular instructions.

A more serious compatibility problem involves pointer parameters. Applications built with the large data memory model may pass pointer parameter values in two words of the stack. But libraries built using the small memory model may expect pointer arguments to occupy a single word of memory. This incompatibility will cause runtime stack corruption.

You may or may not build external libraries or modules written in assembly with extended addressing modes. The linker does not enforce any compatibility rules on assembly language modules or libraries.

The compiler encodes the memory model into the object file. The linker verifies that all objects linked into an executable have compatible memory models. The ELF header's `e_flags` field includes the bit fields that contain the encoded data memory model attributes of the object file:

```
#define EF_M56800E_LDMM 0x00000001 /* Large data memory model  
    flag */
```

Additionally, C language objects are identified by an ELF header flag.

```
#define EF_M56800E_C 0x00000002 /* Object file generated from C  
    source */
```

## Optimizing Code

Register coloring is an optimization specific to DSP56800E development. The compiler assigns two (or more) register variables to the same register, if the code does not use the variables at the same time. The code of Listing 6.6 does not use variables `i` and `j` at the same time, so the compiler could store them in the same register:

### Listing 6.6 Register Coloring Example

---

```
short i;  
int j;
```

---

```
for (i=0; i<100; i++) { MyFunc(i); }
for (j=0; j<100; j++) { MyFunc(j); }
```

---

However, if the code included the expression `MyFunc (i+j)`, the variables would be in use at the same time. The compiler would store the two variables in different registers.

For DSP56800E development, you can instruct the compiler to:

1. **Store all local variables on the stack.** — (That is, do *not* perform register coloring.) The compiler loads and stores local variables when you read them and write to them. You may prefer this behavior during debugging, because it guarantees meaningful values for all variables, from initialization through the end of the function. To have the compiler behave this way, specify optimization Level 0, in the **Global Optimizations** settings panel.
2. **Place as many local variables as possible in registers.** — (That is, *do* perform register coloring.) To have the compiler behave this way, specify optimization Level 1 or higher, in the **Global Optimizations** settings panel.

---

<b>NOTE</b>	Optimization Level 0 is best for code that you will debug after compilation. Other optimization levels include register coloring. If you compile code with an optimization level greater than 0, then debug the code, register coloring could produce unexpected results.
-------------	---

---

Variables declared `volatile` (or those that have the address taken) are not kept in registers and may be useful in the presence of interrupts.

3. **Run Peephole Optimization.** — The compiler eliminates some compare instructions and improves branch sequences. Peephole optimizations are small and local optimizations that eliminate some compare instructions and improve branch sequences. To have the compiler behave this way, specify optimization Levels 1 through 4, in the **Global Optimizations** settings panel.

## Deadstripping and Link Order

The M56800E Linker deadstrips unused code and data only from files compiled by the CodeWarrior C compiler. The linker never deadstrips assembler relocatable files or C object files built by other compilers.

Libraries built with the CodeWarrior C compiler contribute only the used objects to the linked program. If a library has assembly files or files built with other C compilers, the only files that contribute to the linked program are those that have at least one referenced object. If you enable deadstripping, the linker completely ignores files without any referenced objects.

The Link Order page of the project window specifies the order (top to bottom) in which the DSP56800E linker processes C source files, assembly source files, and archive (.a and .lib) files. If both a source-code file and a library file define a symbol, the linker uses the definition of the file that appears first, in the link order. To change the link order, drag the appropriate filename to a different place, in this page's list.

## Working with Peripheral Module Registers

This section highlights the issues and recommends programming style for using bit fields to access memory mapped I/O. Memory mapped I/O is a way of accessing devices that are not on the system. A part of the normal address space is mapped to I/O ports. A read/write to that memory location triggers an access to the I/O device, though to the program it seems like a normal memory access. Even if one byte is written to, in the space allocated to a peripheral register, the whole register is written to. So the other byte of the peripheral register will not retain its data. This may happen because the compiler generates optimal bit-field instructions with a read(byte)-mask-writeback(byte) code sequence.

## Compiler Generates Bit Instructions

The compiler generates BFSET for |=, BFCLR for &=, and BFCHG for ^= operators.

Listing 6.7 shows a C source example and the generated sample code.

### Listing 6.7 C Source Example

---

```
int i;
int *ip;

void main(void)
{
    i &= ~1;
```

# Freescale Semiconductor, Inc.

## C for DSP56800E

### Working with Peripheral Module Registers

---

```
/* generated codes
P: 00000082: 8054022D0001      bfcclr    #1,X:0x00022d
*/

    (*(ip))^= 1;

/* generated codes
P:00000085: F87C022C            moveu.w  X:0x00022c,R0
P:00000087: 84400001            bfchg    #1,X:(R0)
*/

    *((int*)(0x1234))|=1;

/* generated codes
P:00000089: E4081234            move.l   #4660,R0
P:0000008B: 82400001            bfset   #1,X:(R0)
*/

}

/* generated codes
P:0000008D: E708                rts
*/
```

---

Note, the following example:

---

```
#define word int

union {
    word Word;
    struct {
        word SBK :1;
        word RWU :1;
        word RE  :1;
        word TE  :1;
        word REIE :1;
        word RFIE :1;
        word TIIE :1;
        word TEIE :1;
        word PT   :1;
        word PE   :1;
        word POL  :1;
        word WAKE :1;
        word M    :1;
        word RSRC :1;
    };
};
```

## C for DSP56800E

Working with Peripheral Module Registers

---

```
    word SWAI :1;
    word LOOP :1;
} Bits;
} SCICR;

/* Code:*/
    SCICR.Bits.TE = 1;          /* SCICR content is 0x0800 */
    SCICR.Bits.PE = 1;          /* SCICR content is 0x0002 ??? */
```

---

## Explanation of Undesired Behavior

If "SCICR" is mapped to a peripheral register, the code that is used to access the register is not portable and might be unsafe, like in DSP56800E at present.

Bit field behavior in C is almost all implementation defined. So generating the following code is legal:

---

```
    SCICR.Bits.TE = 1;          /* SCICR content is 0x0800 */

/* generated codes
P:00000082:874802c      moveu.w    #SCICR,R0
P:00000084:F0E0000      move.b     X:(R0),A
P:00000086:8350008      bfset      #8,A1
P:00000088:9800         move.b     A1,X:(R0)
*/

    SCICR.Bits.PE = 1;          /* SCICR content is 0x0002 ??? */

/* generated codes
P:00000089:F0E00001     move.b     X:(R0+1),A
P:0000008B:83500002     bfset      #2,A1
P:0000008D:9804         move.b     A1,X:(R0+1)
*/
```

---

However, since the writes (at P:0x88 and at P:0x8D) are byte instructions and only 16 bits can be written to the SCICR register, the other bytes look like they are filled with zero's before the SCICR is overwritten.

The use of byte accesses is due to a compiler optimization that tries to generate the smallest possible memory access.



---

## Recommended Programming Style

The use of a union of a member that can hold the whole register (the "Word" member above) and a struct that can access the bits of the register (the "Bits" member above) is a good idea.

What is recommended is to read the whole memory mapped register (using the "Word" union member) into a local instance of the union, do the bit-manipulation on the local, and then write the result as a whole word in to the memory mapped register. So the C code would look something like:

---

```
#define word int

union SCICR_union{
    word Word;
    struct {
        word SBK    :1;
        word RWU    :1;
        word RE     :1;
        word TE     :1;
        word REIE   :1;
        word RFIE   :1;
        word TIIE   :1;
        word TEIE   :1;
        word PT     :1;
        word PE     :1;
        word POL    :1;
        word WAKE   :1;
        word M      :1;
        word RSRC   :1;
        word SWAI   :1;
        word LOOP   :1;
    } Bits;
} SCICR;

/* Code: */

union SCICR_union localSCICR;
localSCICR.Word = SCICR.Word;

/* generated codes
P:00000083:F07C022C      move.w      X:#SCICR,A
P:00000085:907F         move.w      A1, X: (SP-1)
*/
```

## C for DSP56800E

### Generating MAC Instruction Set

---

```
localSCICR.Bits.TE = 1;

/* generated codes
P:00000086:8AB4FFFF      adda      #-1,SP,R0
P:00000088:F0E00000      move.b    X:(R0),A
P:0000008A:83500008      bfset    #8,A1
P:0000008C:9800          move.b    A1,X: (R0)
*/

localSCICR.Bits.PE = 1;

/* generated codes
P:0000008D:F0E00001      move.b    X: (R0+1),A
P:0000008F:83500002      bfset    #2,A1
P:00000091:9804          move.b    A1,x: (R0+1)
*/

SCICR.Word = localSCICR.Word;

/* generated codes
P:00000092:B67F022C      move.w    X:(SP-1),X:#SCICR
*/
```

---

## Generating MAC Instruction Set

The compiler generates the `imac.l` instruction if the C code performs multiplication on two long operands which are casted to short type; and the product is added to a long type. For example, the following code:

```
short a;
short b;
long c;
....
long d = c+((long)a*(long)b);
....
```

---

generates the following assembly:

```
move.w X:0x000000,Y0 ; Fa
move.w X:0x000000,B ; Fb
move.l X:0x000000,A ; Fc
```

---

---

`imac.l B1,Y0,A`

---

# Freescale Semiconductor, Inc.

**C for DSP56800E**  
*Generating MAC Instruction Set*

---

# High-Speed Simultaneous Transfer

---

High-Speed Simultaneous Transfer (HSST) facilitates data transfer between low-level targets (hardware or simulator) and host-side client applications. The data transfer occurs without stopping the core.

The host-side client must be an IDE plug-in or a script run through the command-line debugger.

When the customer links their application to the target side hsst lib, the debugger detects that the customer wants to use hsst and automatically enables hsst communications.

---

<b>NOTE</b>	To use HSST, you must launch the target side application through the debugger.
-------------	--

---

## Host-Side Client Interface

This section describes the API calls for using High-Speed Simultaneous Transfer (HSST) from your host-side client application.

At the end of this section, an example of a HSST host-side program is given (Listing 7.1 on page 161).

---

### **hsst\_open**

A host-side client application uses this function to open a communication channel with the low-level target. Opening a channel that has already been opened will result in the same channel ID being returned.

## High-Speed Simultaneous Transfer Host-Side Client Interface

---

### Prototype

```
HRESULT hsst_open (  
    const char* channel_name,  
    size_t *cid );
```

### Parameters

channel\_name

Specifies the communication channel name.

cid

Specifies the channel ID associated with the communication channel.

### Returns

S\_OK if the call succeeds or S\_FALSE if the call fails.

---

## hsst\_close

A host-side client application uses this function to close a communication channel with the low-level target.

### Prototype

```
HRESULT hsst_close ( size_t channel_id ) ;
```

### Parameters

channel\_id

Specifies the channel ID of the communication channel to close.

### Returns

S\_OK if the call succeeds or S\_FALSE if the call fails.

---

## hsst\_read

A host-side client application uses this function to read data sent by the target application without stopping the core.

### Prototype

```
HRESULT hsst_read (
    void *data,
    size_t size,
    size_t nmemb,
    size_t channel_id,
    size_t *read );
```

### Parameters

`data`

Specifies the data buffer into which data is read.

`size`

Specifies the size of the individual data elements to read.

`nmemb`

Specifies the number of data elements to read.

`channel_id`

Specifies the channel ID of the communication channel from which to read.

`read`

Contains the number of data elements read.

### Returns

`S_OK` if the call succeeds or `S_FALSE` if the call fails.

## High-Speed Simultaneous Transfer Host-Side Client Interface

---

### hsst\_write

A host-side client application uses this function to write data that the target application can read without stopping the core.

#### Prototype

```
HRESULT hsst_write (  
    void *data,  
    size_t size,  
    size_t nmemb,  
    size_t channel_id,  
    size_t *written );
```

#### Parameters

`data`

Specifies the data buffer that holds the data to write.

`size`

Specifies the size of the individual data elements to write.

`nmemb`

Specifies the number of data elements to write.

`channel_id`

Specifies the channel ID of the communication channel to write to.

`written`

Contains the number of data elements written.

#### Returns

`S_OK` if the call succeeds or `S_FALSE` if the call fails.



---

## hsst\_size

A host-side client application uses this function to determine the size of unread data (in bytes) in the communication channel.

### Prototype

```
HRESULT hsst_size (
    size_t channel_id,
    size_t *unread );
```

### Parameters

channel\_id

Specifies the channel ID of the applicable communication channel.

unread

Contains the size of unread data in the communication channel.

### Returns

S\_OK if the call succeeds or S\_FALSE if the call fails.

---

## hsst\_block\_mode

A host-side client application uses this function to set a communication channel in blocking mode. All calls to read from the specified channel block indefinitely until the requested amount of data is available. By default, a channel starts in the blocking mode.

### Prototype

```
HRESULT hsst_block_mode ( size_t channel_id );
```

### Parameters

channel\_id

Specifies the channel ID of the communication channel to set in blocking mode.

## High-Speed Simultaneous Transfer Host-Side Client Interface

---

### Returns

S\_OK if the call succeeds or S\_FALSE if the call fails.

---

## hsst\_noblock\_mode

A host-side client application uses this function to set a communication channel in non-blocking mode. Calls to read from the specified channel do not block for data availability.

### Prototype

```
HRESULT hsst_noblock_mode ( size_t channel_id );
```

### Parameters

channel\_id

Specifies the channel ID of the communication channel to set in non-blocking mode.

### Returns

S\_OK if the call succeeds or S\_FALSE if the call fails.

---

## hsst\_attach\_listener

Use this function to attach a host-side client application as a listener to a specified communication channel. The client application receives a notification whenever data is available to read from the specified channel.

HSST notifies the client application that data is available to read from the specified channel. The client must implement this function:

```
void NotifiableHSSTClient::Update (size_t descriptor, size_t  
size, size_t nmemb);
```

HSST calls the Notifiable HSST Client:: Update function when data is available to read.

---

## Prototype

```
HRESULT hsst_attach_listener (  
    size_t cid,  
    NotifiableHSSTClient *subscriber );
```

## Parameters

cid

Specifies the channel ID of the communication channel to listen to.

subscriber

Specifies the address of the variable of class `Notifiable HSST Client`.

## Returns

`S_OK` if the call succeeds or `S_FALSE` if the call fails.

---

## hsst\_detach\_listener

Use this function to detach a host-side client application that you previously attached as a listener to the specified communication channel.

## Prototype

```
HRESULT hsst_detach_listener ( size_t cid );
```

## Parameters

cid

Specifies the channel ID of the communication channel from which to detach a previously specified listener.

## Returns

`S_OK` if the call succeeds or `S_FALSE` if the call fails.

### hsst\_set\_log\_dir

A host-side client application uses this function to set a log directory for the specified communication channel.

This function allows the host-side client application to use data logged from a previous High-Speed Simultaneous Transfer (HSST) session rather than reading directly from the board.

After the initial call to `hsst_set_log_dir`, the CodeWarrior software examines the specified directory for logged data associated with the relevant channel instead of communicating with the board to get the data. After all the data has been read from the file, all future reads are read from the board.

To stop reading logged data, the host-side client application calls `hsst_set_log_dir` with `NULL` as its argument. This call only affects host-side reading.

#### Prototype

```
HRESULT hsst_set_log_dir (  
    size_t cid,  
    const char* log_directory );
```

#### Parameters

`cid`

Specifies the channel ID of the communication channel from which to log data.

`log_directory`

Specifies the path to the directory in which to store temporary log files.

#### Returns

`S_OK` if the call succeeds or `S_FALSE` if the call fails.

## HSST Host Program Example

In Listing 7.1 the host is the IDE plugin (DLL) to the interface with the HSST target (DSP56800E) project. This establishes data transfer between the host (your computer) and the target (the DSP56800E board).

---

<b>NOTE</b>	Before launching the program, the IDE plugin needs to be created and placed in the folder: CodeWarrior\bin\Plugins\Com.
-------------	---

---

## Listing 7.1 Sample HSST Host Program

```
#include "CodeWarriorCommands.h"
#include "HSSTInterface.h"
#include <stdio>
#include <stdlib>

unsigned __stdcall HSSTClientMain ( void *pArguments );

#define buf_size 1000                                /* Data size */

/* Assigning name for Plugin and Menu Title */
extern const CWPluginID kToolbarTestPluginID = "HSST_host_sample";
extern const wchar_t* MenuTitle = L"HSST_host_sample";

unsigned __stdcall HSSTClientMain ( void *pArguments )
{
    IMWHSST_Client *pHSST = (IMWHSST_Client *)pArguments;

    long data[buf_size];
    size_t channel_1, channel_2, read_items, written_items;

    /* Opening channel 1 and 2 from HOST side */
    HRESULT hr_1 = pHSST->hsst_open ( "channel_1",
        &channel_1 );
    HRESULT hr_2 = pHSST->hsst_open ( "channel_2",
        &channel_2 );

    /* HOST reading data from channel 1 */
    pHSST->hsst_read ( data, sizeof(long), buf_size, channel_1,
        &read_items );

    /* HOST writing data to channel 2 */
    pHSST->hsst_write( data, sizeof(long), buf_size, channel_2,
        &written_items );
}
```

## High-Speed Simultaneous Transfer Target Library Interface

---

```
    return 0;  
}
```

---

## Target Library Interface

This section describes the API calls for using High-Speed Simultaneous Transfer (HSST) from your target application.

At the end of this section, an example of a HSST target program is given (Listing 7.2 on page 169).

---

### HSST\_open

A target application uses this function to open a bidirectional communication channel with the host. The default setting is for the function to open an output channel in buffered mode. Opening a channel that has already been opened will result in the same channel ID being returned.

#### Prototype

```
HSST_STREAM* HSST_open ( const char *stream );
```

#### Parameters

stream

Passes the communication channel name.

#### Returns

The stream associated with the opened channel.

---

### HSST\_close

A target application uses this function to close a communication channel with the host.

## Prototype

```
int    HSST_close ( HSST_STREAM *stream );
```

## Parameters

stream

Passes a pointer to the communication channel.

## Returns

0 if the call was successful or -1 if the call was unsuccessful.

---

## HSST\_setvbuf

A target application can use this function to perform the following actions:

- Set an open channel opened in write mode to use buffered mode

---

<b>NOTE</b>	This can greatly improve performance.
-------------	---------------------------------------

---

- Resize the buffer in an existing buffered channel opened in write mode
- Provide an external buffer for an existing channel opened in write mode
- Reset buffering to unbuffered mode

You can use this function only after you successfully open the channel.

The contents of a buffer (either internal or external) at any time are indeterminate.

## Prototype

```
int    HSST_setvbuf (
        HSST_STREAM *rs,
        unsigned char *buf,
        int mode,
        size_t size );
```

## Parameters

rs

Specifies a pointer to the communication channel.

## High-Speed Simultaneous Transfer Target Library Interface

---

`buf`

Passes a pointer to an external buffer.

`mode`

Passes the buffering mode as either buffered (specified as HSSTFBUF) or unbuffered (specified as HSSTNBUF).

`size`

Passes the size of the buffer.

### Returns

0 if the call was successful or -1 if the call was unsuccessful.

---

### NOTE

You must flush the buffers before exiting the program to ensure that all the data that has been written is sent to the host. For more details, see HSST\_flush.

---

---

## HSST\_write

A target application uses this function to write data for the host-side client application to read.

### Prototype

```
size_t HSST_write (  
    void *data,  
    size_t size,  
    size_t nmem,  
    HSST_STREAM *stream );
```

### Parameters

`data`

Passes a pointer to the data buffer holding the data to write.

`size`

Passes the size of the individual data elements to write.



`nmemb`

Passes the number of data elements to write.

`stream`

Passes a pointer to the communication channel.

### Returns

The number of data elements written.

---

## HSST\_read

A target application uses this function to read data sent by the host.

### Prototype

```
size_t HSST_read (
    void *data,
    size_t size,
    size_t nmemb,
    HSST_STREAM *stream );
```

### Parameters

`data`

Passes a pointer to the data buffer into which to read the data.

`size`

Passes the size of the individual data elements to read.

`nmemb`

Passes the number of data elements to read.

`stream`

Passes a pointer to the communication channel.

### Returns

The number of data elements read.

## High-Speed Simultaneous Transfer Target Library Interface

---

### HSST\_flush

A target application uses this function to flush out data buffered in a buffered output channel.

#### Prototype

```
int HSST_flush ( HSST_STREAM *stream );
```

#### Parameters

*stream*

Passes a pointer to the communication channel. The High-Speed Simultaneous Transfer (HSST) feature flushes all open buffered communication channels if this parameter is null.

#### Returns

0 if the call was successful or -1 if the call was unsuccessful.

---

### HSST\_size

A target application uses this function to determine the size of unread data (in bytes) for the specified communication channel.

#### Prototype

```
size_t HSST_size ( HSST_STREAM *stream );
```

#### Parameters

*stream*

Passes a pointer to the communication channel.

#### Returns

The number of bytes of unread data.

---

## HSST\_raw\_read

A target application uses this function to read raw data from a communication channel (without any automatic conversion for endianness while communicating).

### Prototype

```
size_t    HSST_raw_read (
    void *ptr,
    size_t length,
    HSST_STREAM *rs );
```

### Parameters

*ptr*

Specifies the pointer that points to the buffer into which data is read.

*length*

Specifies the size of the buffer in bytes.

*rs*

Specifies a pointer to the communication channel.

### Returns

The number of bytes of raw data read.

---

<b>NOTE</b>	This function is useful for sending data structures (e.g., C-type structures).
-------------	--

---

---

## HSST\_raw\_write

A target application uses this function to write raw data to a communication channel (without any automatic conversion for endianness while communicating).

## High-Speed Simultaneous Transfer Target Library Interface

---

### Prototype

```
size_t    HSST_raw_write (
    void *ptr,
    size_t length,
    HSST_STREAM *rs );
```

### Parameters

ptr

Specifies the pointer that points to the buffer that holds the data to write.

length

Specifies the size of the buffer in bytes.

rs

Specifies a pointer to the communication channel.

### Returns

The number of data elements written.

---

<b>NOTE</b>	This function is useful for sending data structures (e.g., C-type structures).
-------------	--

---

---

## HSST\_set\_log\_dir

A target application uses this function to set the host-side directory for storing temporary log files. Old logs that existed prior to the call to `HSST_set_log_dir()` are over-written. Logging stops when the channel is closed or when `HSST_set_log_dir()` is called with a null argument. These logs can be used by the host-side function `HSST_set_log_dir`.

### Prototype

```
int    HSST_set_log_dir (
    HSST_STREAM *stream,
    char *dir_name );
```

---

## Parameters

stream

Passes a pointer to the communication channel.

dir\_name

Passes a pointer to the path to the directory in which to store temporary log files.

## Returns

0 if the call was successful or -1 if the call was unsuccessful.

## HSST Target Program Example

In Listing 7.2 the HSST target program runs in parallel with the host plugin. The target communicates with the host-side (your computer).

---

<b>NOTE</b>	To restart the program after execution, click on <b>Restart HSST</b> as shown in Figure 7.1.
-------------	--

---

### Listing 7.2 Sample HSST Target Program

---

```
#include <stdio.h>
#include <stdlib.h>
#include "HSST.h"

#define buf_size 1000      /* Data size */

long i, test_buffer[buf_size];

int main ( )
{
    HSST_STREAM *channel_1, *channel_2;
    int written_items=0;
    int read_items=0;

    for ( i = 0; i < buf_size; ++ i )
    {
        test_buffer[i] = i;
    }
}
```

## High-Speed Simultaneous Transfer Target Library Interface

---

```
/* Opening channel 1 and 2 from TARGET side */
channel_1 = HSST_open ( "channel_1" );
channel_2 = HSST_open ( "channel_2" );

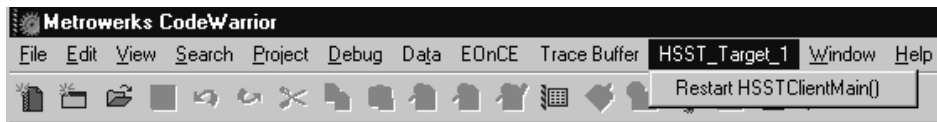
/* TARGET writing data to channel 1 */
written_items = HSST_write(test_buffer, sizeof(long),
    buf_size, channel_1);

/* TARGET reading data from channel 2 */
read_items = HSST_read(test_buffer, sizeof(long), buf_size,
    channel_2);

return 0;
}
```

---

**Figure 7.1 Restart HSST**



---

<b>NOTE</b>	For an HSST example, see the HSST example in this path: {CodeWarrior path} (CodeWarrior_Examples) \ DSP56800E_hsst_client-to-client
-------------	---

---

# Data Visualization

---

Data visualization lets you graph variables, registers, regions of memory, and HSST data streams as they change over time.

The Data Visualization tools can plot memory data, register data, global variable data, and HSST data.

- Starting Data Visualization
- Data Target Dialog Boxes
- Graph Window Properties

## Starting Data Visualization

To start the Data Visualization tool:

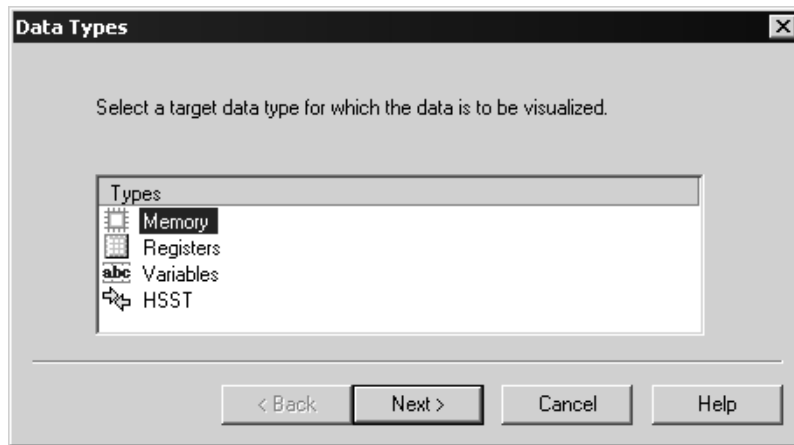
1. Start a debug session
2. Select Data Visualization > Configurator.

The Data Types window (Figure 8.1) appears. Select a data target type and click the Next button.

## Data Visualization

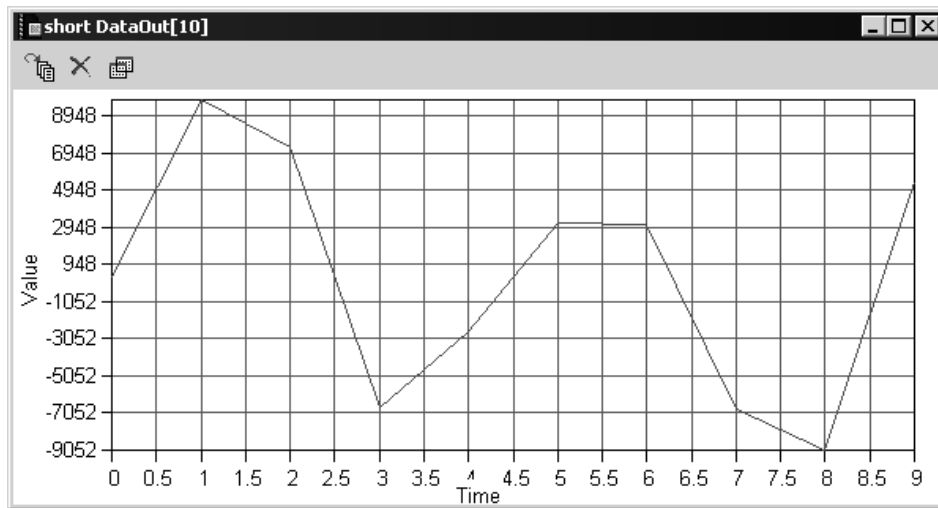
### Starting Data Visualization

**Figure 8.1 Data Types Window**



3. Configure the data target dialog box and filter dialog box.
4. Run your program to display the data (Figure 8.2).

**Figure 8.2 Graph Window**





## Data Target Dialog Boxes

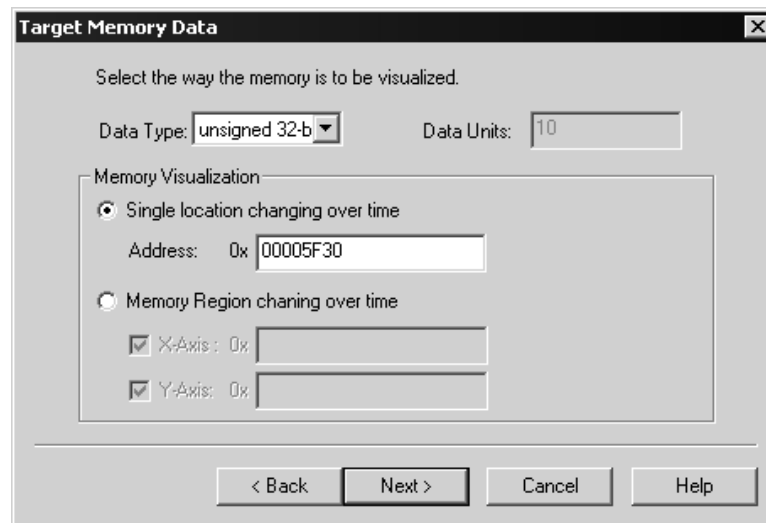
There are four possible data targets. Each target has its own configuration dialog.

- Memory
- Registers
- Variables
- HSST

### Memory

The Target Memory dialog box lets you graph memory contents in real-time.

**Figure 8.3 Target Memory Dialog Box**



## Data Type

The Data Type list box lets you select the type of data to be plotted.

## Data Unit

The Data Units text field lets you enter a value for number of data units to be plotted. This option is only available when you select Memory Region Changing Over Time.

## Single Location Changing Over Time

The Single Location Changing Over Time option lets you graph the value of a single memory address. Enter this memory address in the Address text field.

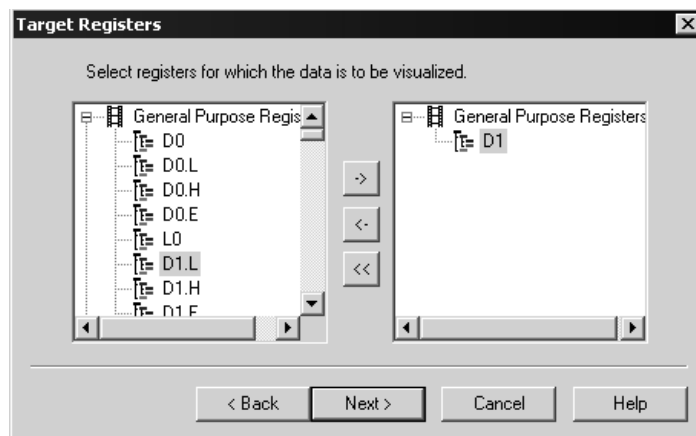
## Memory Region Changing Over Time

The Memory Region Changing Over Time options lets you graph the values of a memory region. Enter the memory addresses for the region in the X-Axis and Y-Axis text fields.

## Registers

The Target Registers dialog box lets you graph the value of registers in real-time.

**Figure 8.4 Target Registers Dialog Box**

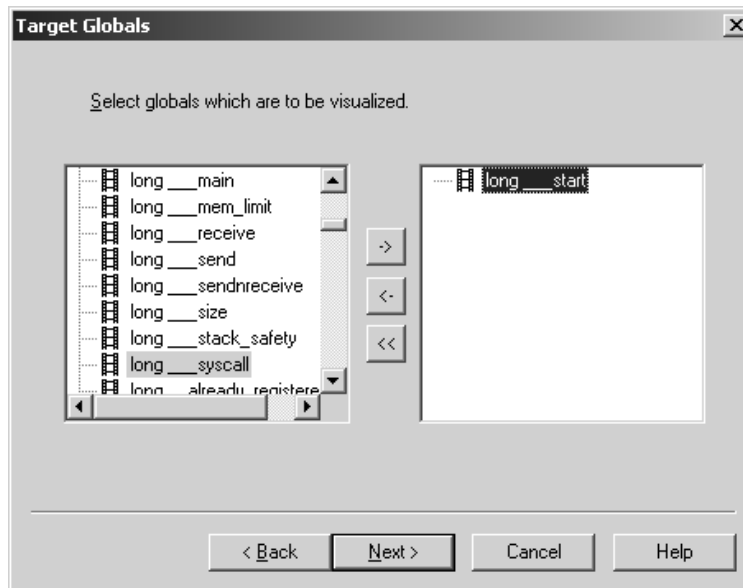


Select registers from the left column, and click the -> button to add them to the list of registers to be plotted.

## Variables

The Target Globals dialog box lets you graph the value of global variables in real-time. (See Figure 8.5.)

**Figure 8.5 Target Globals Dialog Box**



Select global variables from the left column, and click the -> button to add them to the list of variables to be plotted.

## HSST

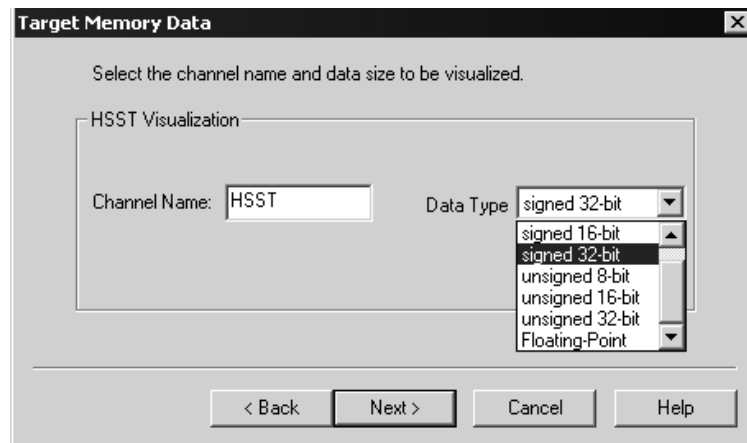
The Target HSST dialog box lets you graph the value of an HSST stream in real-time. (See Figure 8.6.)

---

**NOTE** To plot HSST data, the data visualization tool needs its own HSST channel. Make sure your program opens a separate channel exclusively for the data visualization window. This will avoid impacting data transmissions on other channels.

---

**Figure 8.6 Target HSST Dialog Box**



### Channel Name

The Channel Name text field lets you specify the name of the HSST stream to be plotted.

### Data Type

The Data Type list box lets you select the type of data to be plotted.

## Graph Window Properties


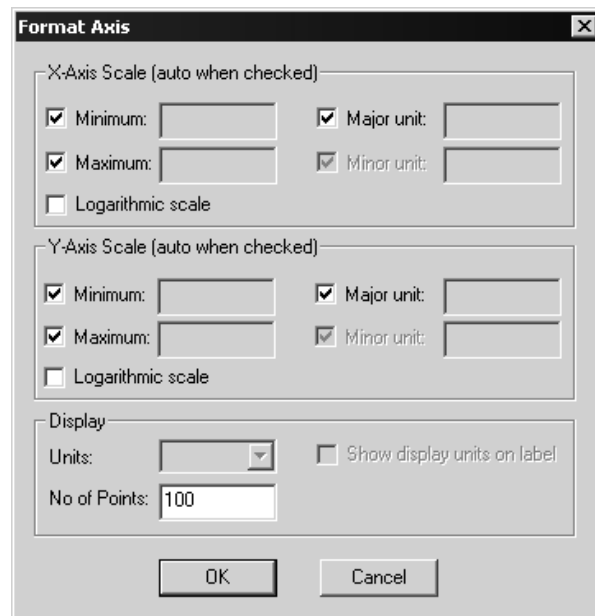
To change the look of the graph window, click the  graph properties button to open the Format Axis dialog box.

Figure 8.7 Format Axis Dialog Box



### Scaling

The default scaling settings of the data visualization tools automatically scale the graph window to fit the existing data points.

To override the automatic scaling, uncheck a scaling checkbox to enable the text field and enter your own value.

To scale either axis logarithmically, enable the Logarithmic Scale option of the corresponding axis.

## Display

The Display settings let you change the maximum number of data points that are plotted on the graph.

---

<b>NOTE</b>	For a data visualization example that uses HSST, see the data visualization example in this path: <code>{CodeWarrior path}\(CodeWarrior_Examples)\hsst_Data_Visualization</code>
-------------	---

---

# Debugging for DSP56800E

---

This chapter, which explains the generic features of the CodeWarrior™ debugger, consists of these sections:

- Target Settings for Debugging
- Command Converter Server
- Launching and Operating the Debugger
- Load/Save Memory
- Fill Memory
- Save/Restore Registers
- EOnCE Debugger Features
- Using the DSP56800E Simulator
- Register Details Window
- Loading a .elf File without a Project
- Using the Command Window
- System-Level Connect
- Debugging in the Flash Memory
- Notes for Debugging on Hardware

## Target Settings for Debugging

This section explains how to control the debugger by modifying the appropriate settings panels.

To properly debug DSP56800E software, you must set certain preferences in the **Target Settings** window. The **M56800E Target** panel is specific to DSP56800E development. The remaining settings panels are generic to all build targets.

Other settings panels can affect debugging. Table 9.1 lists these panels.

**Table 9.1 Setting Panels that Affect Debugging**

This panel...	Affects...	Refer to...
M56800E Linker	symbolics, linker warnings	"Deadstripping and Link Order"
M56800E Processor	optimizations	"Optimizing Code"
Debugger Settings	Debugging options	
Remote Debugging	Debugging communication protocol	"Remote Debugging"
Remote Debug Options	Debugging options	"Remote Debug Options"

The **M56800E Target** panel is unique to DSP56800E debugging. The available options in this panel depend on the DSP56800E hardware you are using and are described in detail in the section on "Remote Debug Options".

## Command Converter Server

The command converter server (CCS) handles communication between the CodeWarrior debugger and the target board. An icon in the status bar indicates the CCS is running. The CCS is automatically launched by your project when you start a CCS debug session if you are debugging a target board using a local machine. However, when debugging a target board connected to a remote machine, see "Setting Up a Remote Connection" on page 185.

---

**NOTE** Projects are set to debug locally by default. The protocol the debugger uses to communicate with the target board, for example, PCI, is determined by how you installed the CodeWarrior software. To modify the protocol, make changes in the **Metrowerks Command Converter Server** window (Figure 9.3).

---



---

## Essential Target Settings for Command Converter Server

Before you can download programs to a target board for debugging, you must specify the target settings for the command converter server:

- Local Settings

If you specify that the CodeWarrior IDE start the command converter server locally, the command converter server uses the connection port (for example, LPT1) that you specified when you installed CodeWarrior Development Studio for Freescale 56800/E Hybrid Controllers.

- Remote Settings

If you specify that the CodeWarrior IDE start the command converter server on a remote machine, specify the IP address of the remote machine on your network (as described in “Setting Up a Remote Connection” on page 185.)

- Default Settings

By default, the command converter server listens on port 41475. You can specify a different port number for the debugger to connect to if needed (as described in “Setting Up a Remote Connection” on page 185.) This is necessary if the CCS is configured to a port other than 41475.

After you have specified the correct settings for the command converter server (or verified that the default settings are correct), you can download programs to a target board for debugging.

The CodeWarrior IDE starts the command converter server at the appropriate time if you are debugging on a local target.

Before debugging on a board connected to a remote machine, ensure the following:

- The command converter server is running on the remote host machine.
- Nobody is debugging the board connected to the remote host machine.

## Changing the Command Converter Server Protocol to Parallel Port

If you specified the wrong parallel port for the command converter server when you installed CodeWarrior Development Studio for Freescale 56800/E Hybrid Controllers, you can change the port.

Change the parallel port:

## Debugging for DSP56800E Command Converter Server

---

1. Click the command converter server icon.

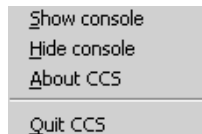
While the command converter server is running, locate the command converter server icon on the status bar. Right-click on the command converter server icon (Figure 9.1):

**Figure 9.1 Command Converter Server Icon**



A menu appears (Figure 9.2):

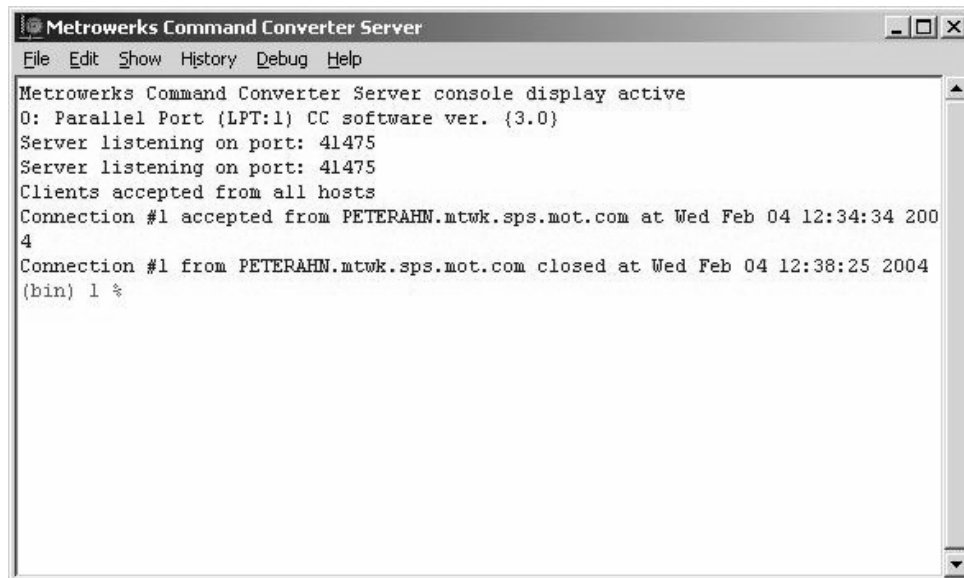
**Figure 9.2 Command Converter Server Menu**



2. Select Show console from the menu.

The **Metrowerks Command Converter Server** window appears (Figure 9.3).

**Figure 9.3 Metrowerks Command Converter Server Window**



3. On the console command line, type the following command:  
`delete all`
4. Press Enter.
5. Type the following command, substituting the number of the parallel port to use (for example, 1 for LPT1):  
`config cc parallel:1`
6. Press Enter.
7. Type the following command to save the configuration:  
`config save`
8. Press Enter.

## Changing the Command Converter Server Protocol to HTI

To change the command converter server to an HTI Connection:

1. While the command converter server is running, right-click on the command converter server icon shown in Figure 9.1 or double click on it.
2. From the menu shown in Figure 9.2, select Show Console.
3. At the console command line in the Metrowerks Command Converter Server window shown in Figure 9.3, type the following command:  

```
delete all
```
4. Press Enter.
5. Type the following command:  

```
config cc: address
```

(substituting for **address** the name of the IP address of your CodeWarrior HTI)

---

<b>NOTE</b>	If the software rejects this command, your CodeWarrior HTI may be an earlier version. Try instead the command: <code>config cc nhti:address</code> , or the command: <code>config cc Panther:address</code> , substituting for <b>address</b> the IP address of the HTI.
-------------	--

---

6. Press Enter.
7. Type the following command to save the configuration:  

```
config save
```
8. Press Enter.

## Changing the Command Converter Server Protocol to PCI

To change the command converter server to a PCI Connection:

1. While the command converter server is running, right-click on the command converter server icon shown in Figure 9.1 or double click on it.
2. From the menu shown in Figure 9.2, select Show Console.
3. At the console command line in the Metrowerks Command Converter Server window shown in Figure 9.3, type the following command:  

```
delete all
```
4. Press Enter.
5. Type the following command:  

```
config cc pci
```
6. Press Enter.
7. Type the following command to save the configuration:  

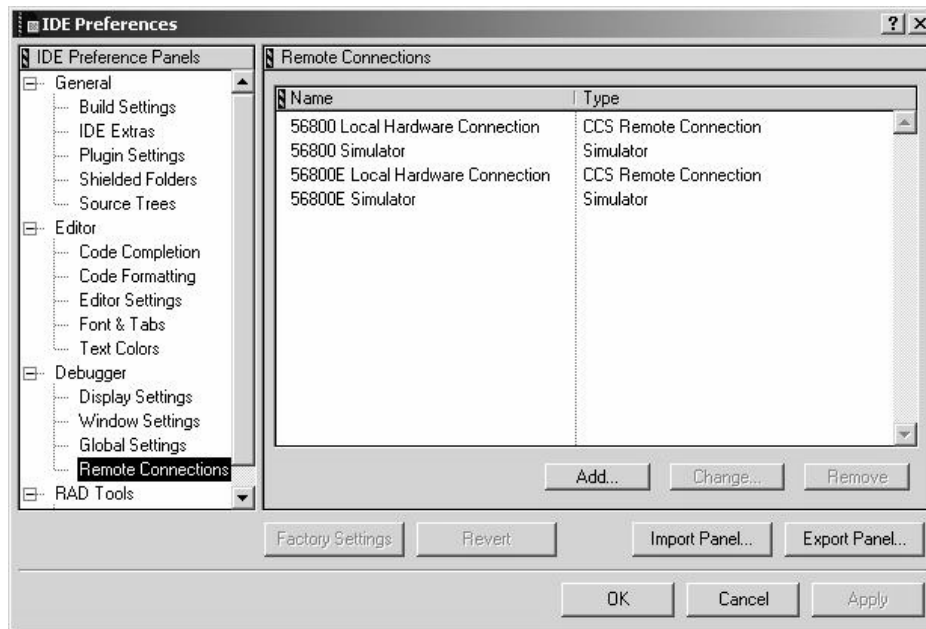
```
config save
```
8. Press Enter.

## Setting Up a Remote Connection

A remote connection is a type of connection to use for debugging along with any preferences that connection may need. To change the preferences for a remote connection or to create a new remote connection:

1. On the main menu, select Edit > Preferences.  
The IDE Preferences Window appears.
2. Click Remote Connections in the left column.  
The **Remote Connections** panel shown in Figure 9.4 appears.

Figure 9.4 Remote Connections Panel



## To Add a New Remote Connection

To add a new remote connection:

1. Click the Add button.

The **New Connection** window appears as shown in Figure 9.5.

**Figure 9.5 New Connection Window**

2. In the Name edit box, type in the connection name.
3. Check Use Remote CCS checkbox.  
Select this checkbox to specify that the CodeWarrior IDE is connected to a remote command converter server. Otherwise, the IDE starts the command converter server locally
4. Enter the Server IP address or host machine name.  
Use this text box to specify the IP address where the command converter server resides when running the command converter server from a location on the network.
5. Enter the Port # to which the command converter server listens or use the default port, which is 41475.

6. Click the OK button.

## To Change an Existing Remote Connection

To change an existing remote connection:

Double click on the connection name that you want to change, or click once on the connection name and click the **Change** button (shown in Figure 9.4 in grey).

## To Remove an Existing Remote Connection

To remove an existing remote connection:

Click once on the connection name and click the **Remove** button (shown in Figure 9.4 in grey).

## Debugging a Remote Target Board

For debugging a target board connected to a remote machine with Code Warrior IDE installed, perform the following steps:

1. Connect the target board to the remote machine.
2. Launch the command converter server (CCS) on the remote machine with the local settings configuration using instructions described in the section “Essential Target Settings for Command Converter Server” on page 181.
3. In the Target Settings>Remote Debugging panel for your project, make sure the proper remote connection is selected.
4. Launch the debugger.

## Launching and Operating the Debugger

---

<b>NOTE</b>	CodeWarrior IDE automatically enables the debugger and sets debugger-related settings within the project.
-------------	---

---

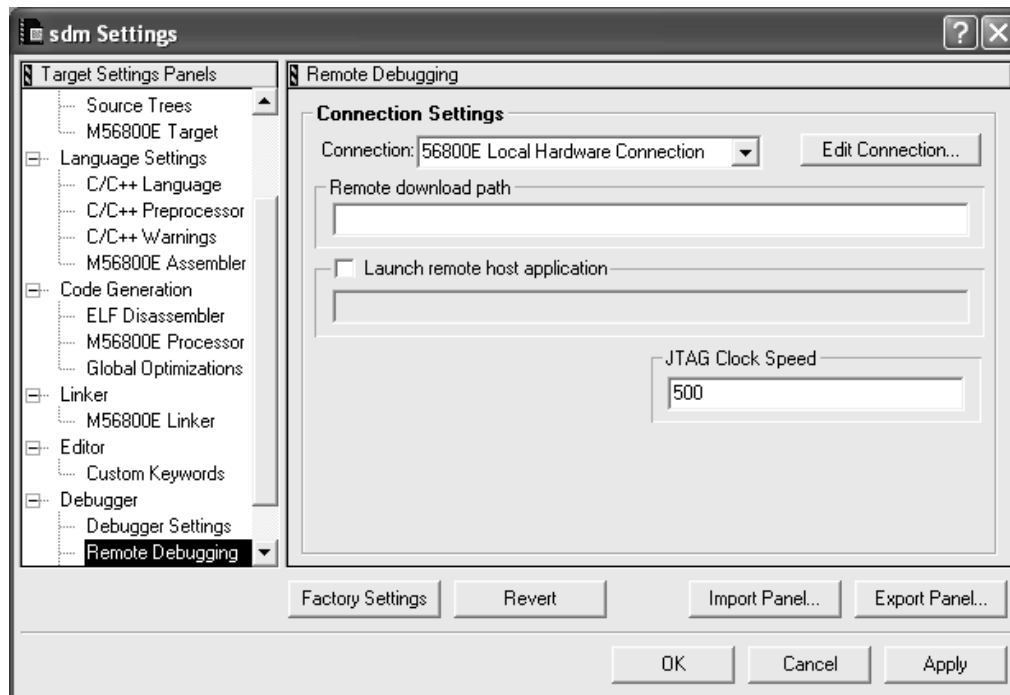


1. Set debugger preferences.

Select **Edit > sdm Settings** from the menu bar of the Metrowerks CodeWarrior window.

The IDE displays the **Remote Debugging** window.

**Figure 9.6 Remote Debugging Panel**



2. Select the Connection.

For example, select **56800E Local Hardware Connection (CCS)**.

3. Click OK button.

4. Debug the project.

Use either of the following options:

- From the Metrowerks CodeWarrior window, select **Project > Debug**.
- Click the **Debug** button in the project window.

## Debugging for DSP56800E

### *Launching and Operating the Debugger*

---

This command resets the board (if **Always reset on download** is checked in the Debugger's **M56800E Target** panel shown in Figure 4.13) and the download process begins.

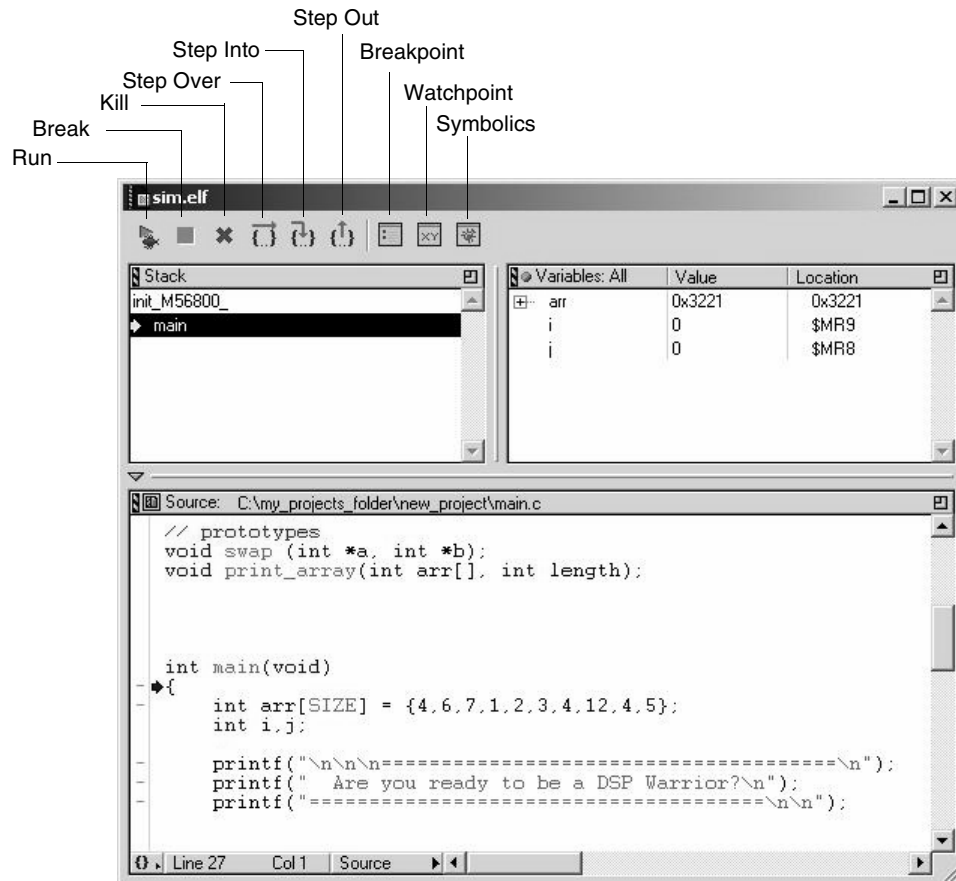
When the download to the board is complete, the IDE displays the **Program** window (**sdm.elf** in sample) shown in Figure 9.7.

---

<b>NOTE</b>	Source code is shown only for files that are in the project folder or that have been added to the project in the project manager, and for which the IDE has created debug information. You must navigate the file system in order to locate sources that are outside the project folder and not in the project manager, such as library source files.
-------------	---

---

**Figure 9.7 Program Window**



## 5. Navigate through your code.

The **Program** window has three panes:

- Stack pane

The **Stack** pane shows the function calling stack.

- Variables pane

The **Variables** pane displays local variables.

- Source pane

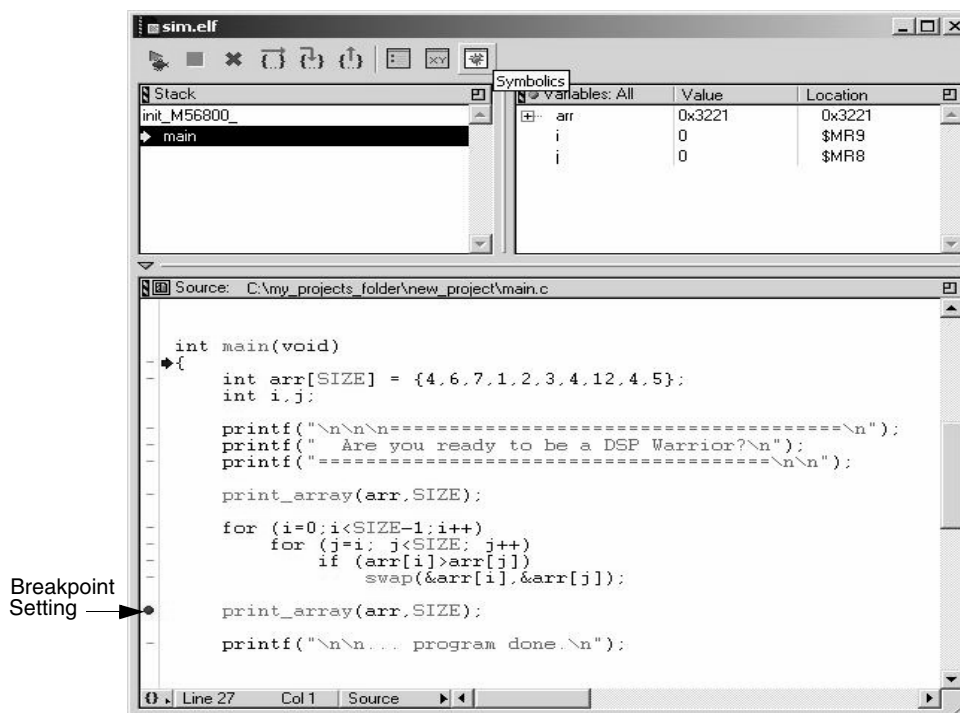
The **Source** pane displays source or assembly code.

The toolbar at the top of the window has buttons that allows you access to the execution commands in the **Debug** menu.

## Setting Breakpoints and Watchpoints

1. Locate the code line.  
Scroll through the code in the **Source** pane of the **Program** window until you come across the `main()` function.
2. Select the code line.  
Click the gray dash in the far left-hand column of the window, next to the first line of code in the `main()` function. A red dot appears (Figure 9.8), confirming you have set your breakpoint.

**Figure 9.8 Breakpoint in the Program Window**



---

<b>NOTE</b>	To remove the breakpoint, click the red dot. The red dot disappears.
-------------	--

---

For more details on how to set breakpoints and use watchpoints, see the *CodeWarrior IDE User's Guide*.

---

<b>NOTE</b>	For the DSP56800E only one watchpoint is available. This watchpoint is only available on hardware targets.
-------------	--

---

## Viewing and Editing Register Values

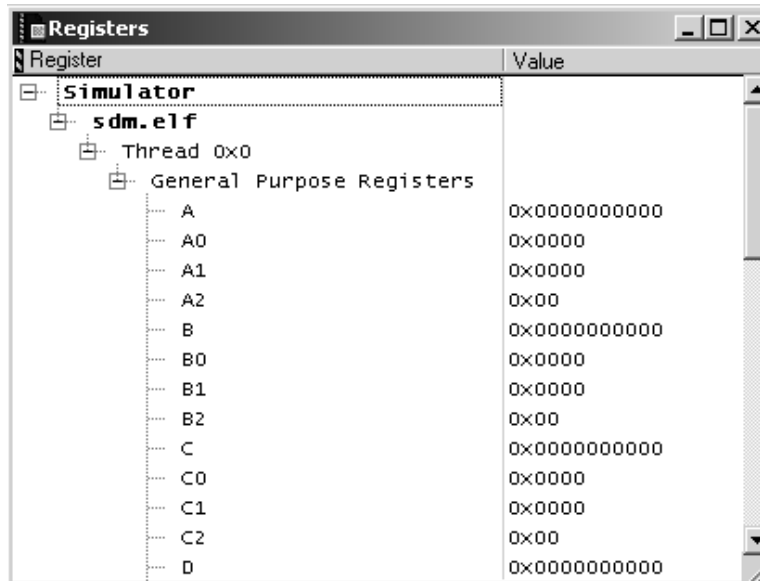
Registers are platform-specific. Different chip architectures have different registers.

1. **Access the Registers** window.

From the menu bar of the Metrowerks CodeWarrior window, select **View > Registers**.

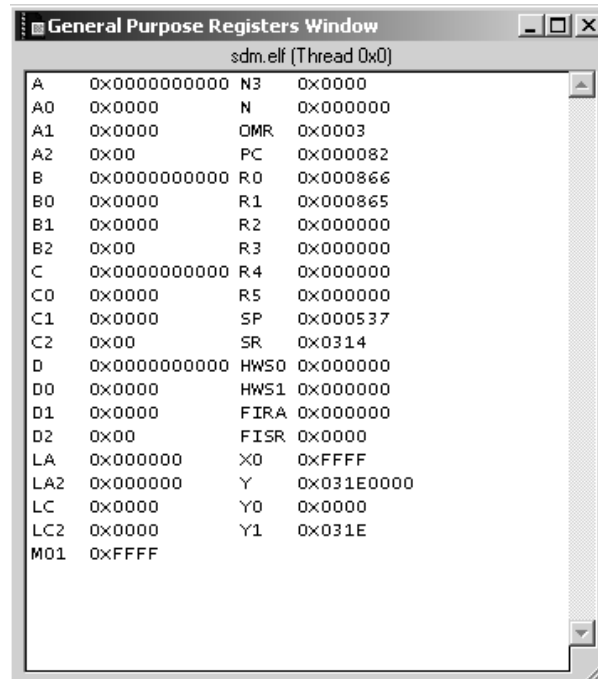
Expand the **General Purpose Registers** tree control to view the registers as in Figure 9.9, or double-click on **General Purpose Registers** to view the registers as in Figure 9.10.

**Figure 9.9 General Purpose Registers for DSP56800E**



Register	Value
<b>Simulator</b>	
sdm.elf	
Thread 0x0	
General Purpose Registers	
A	0x00000000
A0	0x0000
A1	0x0000
A2	0x00
B	0x00000000
B0	0x0000
B1	0x0000
B2	0x00
C	0x00000000
C0	0x0000
C1	0x0000
C2	0x00
D	0x00000000

**Figure 9.10 General Purpose Registers Window**



2. Edit register values.

To edit values in the register window, double-click a register value. Change the value as you wish.

3. Exit the window.

The modified register values are saved.

**NOTE** To view peripheral registers, select the appropriate processor form the processor list box in the M56800E Target Settings Panel.

## Viewing X: Memory

You can view X memory space values as hexadecimal values with ASCII equivalents. You can edit these values at debug time.

---

On targets that have Flash ROM, you cannot edit those values in the memory window that reside in Flash memory.

---

1. Locate a particular address in program memory.

From the menu bar of the Metrowerks CodeWarrior window, select **Data > View Memory**.

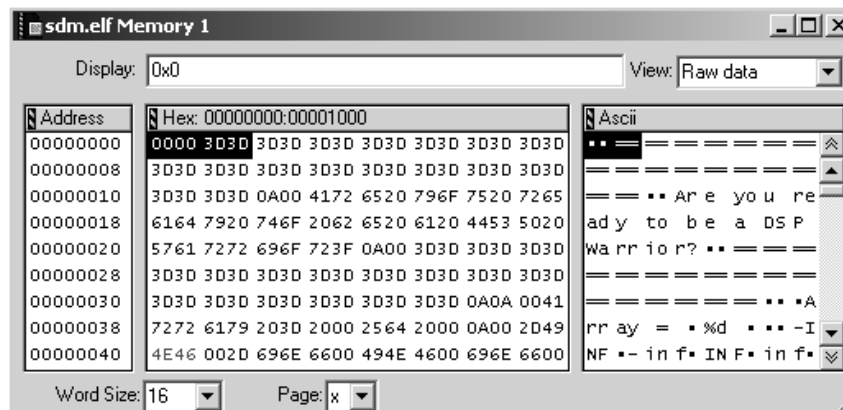
---

**NOTE** The **Source** pane in the **Program** window needs to be the active one in order for the **Data > View Memory** to be activated.

---

The **Memory** window appears (Figure 9.11).

**Figure 9.11 View X:Memory Window**



2. Select type of memory.

Locate the **Page** list box at the bottom of the **View Memory** window. Select **X** for X Memory.

3. Enter memory address.

Type the memory address in the **Display** field located at the top of the **Memory** window.

To enter a hexadecimal address, use standard C hex notation, for example, 0x0.



---

<b>NOTE</b>	You also can enter the symbolic name whose value you want to view by typing its name in the <b>Display</b> field of the <b>Memory</b> window.
-------------	---

---

---

<b>NOTE</b>	The other view options (Disassembly, Source and Mixed) do not apply when viewing X memory.
-------------	--

---

## Viewing P: Memory

You can view P memory space and edit the opcode hexadecimal values at debug time.

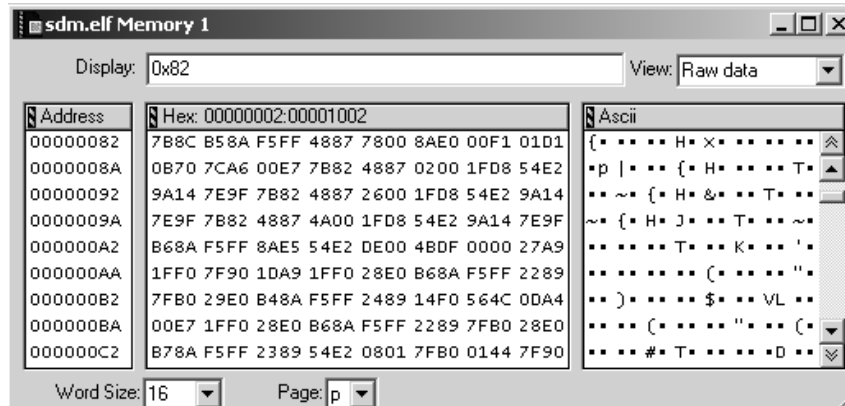
---

<b>NOTE</b>	On targets that have Flash ROM, you cannot edit those values in the memory window that reside in Flash memory.
-------------	--

---

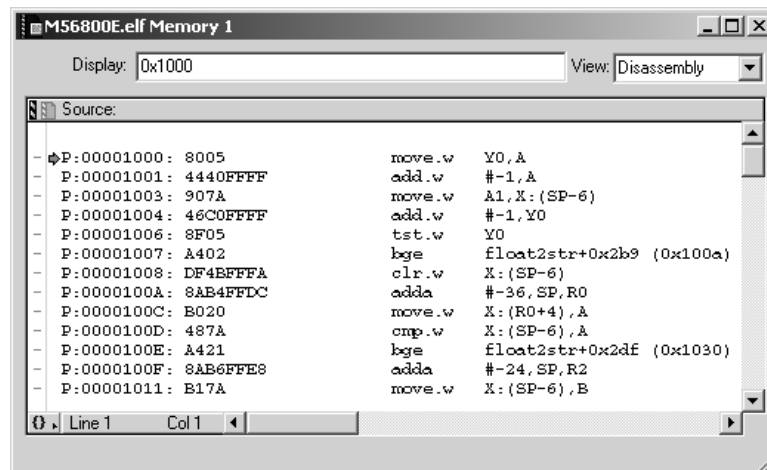
1. Locate a particular address in program memory.  
To view program memory, from the menu bar of the Metrowerks CodeWarrior window, select **Data > View Memory**.  
The **Memory** window appears (Figure 9.11).
2. Select type of memory.  
Locate the **Page** list box at the bottom of the **View Memory** window. Select **P** for P Memory.
3. Enter memory address.  
Type the memory address in the **Display** field located at the top of the **Memory** window.  
To enter a hexadecimal address, use standard C hex notation, for example: 0x82.
4. Select how you want to view P memory.  
Using the **View** list box, you have the option to view P Memory in four different ways.
  - **Raw Data** (Figure 9.12).

Figure 9.12 View P:Memory (Raw Data) Window



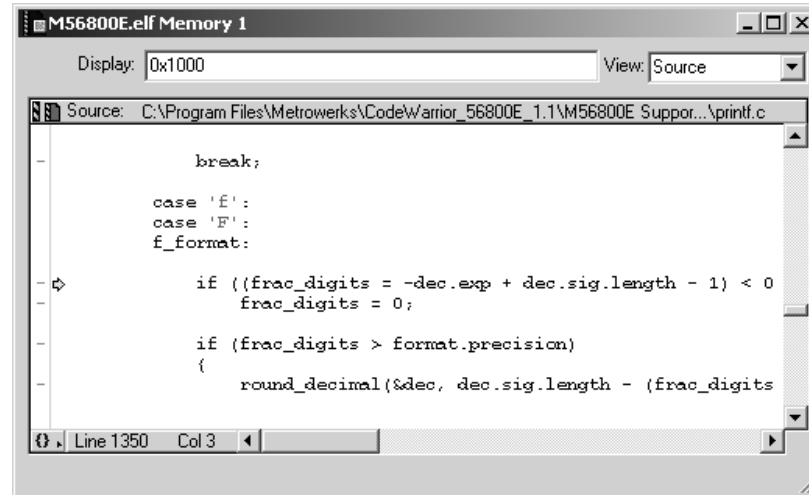
- Disassembly (Figure 9.13).

Figure 9.13 View P:Memory (Disassembly) Window



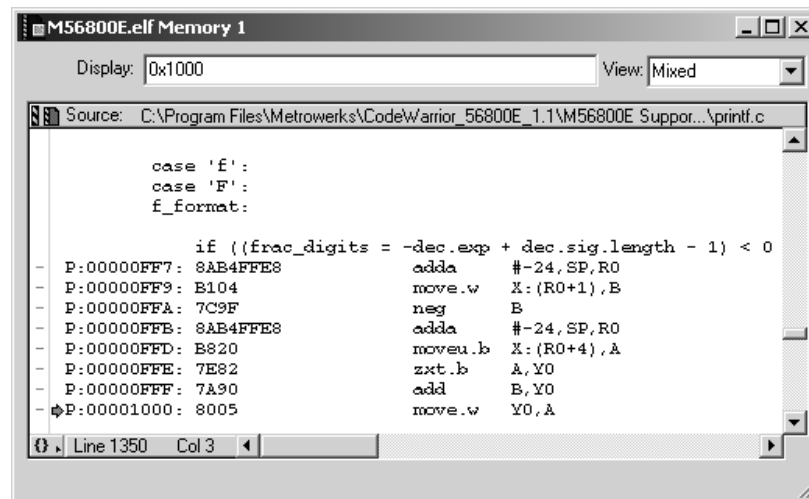
- Source (Figure 9.14).

**Figure 9.14 View P:Memory (Source) Window**



- **Mixed** (Figure 9.15).

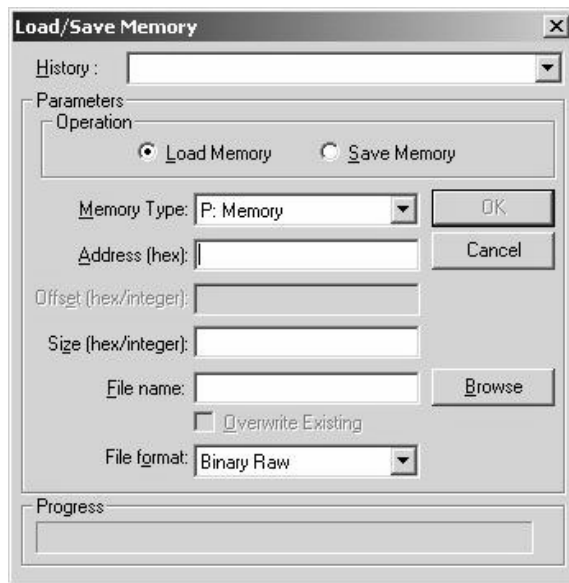
**Figure 9.15 View P:Memory (Mixed) Window**



## Load/Save Memory

From the menu bar of the Metrowerks CodeWarrior window, select **Debug > 56800E > Load/Save Memory** to display the **Load/Save Memory** dialog box (Figure 9.16).

Figure 9.16 Load/Save Memory Dialog Box



Use this dialog box to load and save memory at a specified location and size with a user-specified file. You can associate a key binding with this dialog box for quick access. Press the **Tab** key to cycle through the dialog box displays, which lets you quickly make changes without using the mouse.

### History Combo Box

The **History** combo box displays a list of recent loads and saves. If this is the first time you load or save, the **History** combo box is empty. If you load/save more than once, the combo box fills with the memory address of the start of the load or save and the size of the fill, to a maximum of ten sessions.

If you enter information for an item that already exists in the history list, that item moves up to the top of the list. If you perform another operation, that item appears first.

---

<b>NOTE</b>	By default, the <b>History</b> combo box displays the most recent settings on subsequent viewings.
-------------	--

---

## Radio Buttons

The **Load/Save Memory** dialog box has two radio buttons:

- Load Memory
- Save Memory

The default is **Load Memory**.

## Memory Type Combo Box

The memory types that appear in the **Memory Type** Combo box are:

- P: Memory (Program Memory)
- X: Memory (Data Memory)

## Address Text Field

Specify the address where you want to write the memory. If you want your entry to be interpreted as hex, prefix it with 0x; otherwise, it is interpreted as decimal.

## Size Text Field

Specify the number of words to write to the target. If you want your entry to be interpreted as hex, prefix it with 0x; otherwise, it is interpreted as decimal.

## Dialog Box Controls

### Cancel, Esc, and OK

In Load and Save operations, all controls are disabled except **Cancel** for the duration of the load or save. The status field is updated with the current progress of the operation. Clicking **Cancel** halts the operation, and re-enables the controls on the dialog box. Clicking **Cancel** again closes the dialog box. Pressing the **Esc** key is same as clicking the **Cancel** button.

## Debugging for DSP56800E

### Fill Memory

---

With the **Load Memory** radio button selected, clicking **OK** loads the memory from the specified file and writes it to memory until the end of the file or the size specified is reached. If the file does not exist, an error message appears.

With the **Save Memory** radio button selected, clicking **OK** reads the memory from the target piece by piece and writes it to the specified file. The status field is updated with the current progress of the operation.

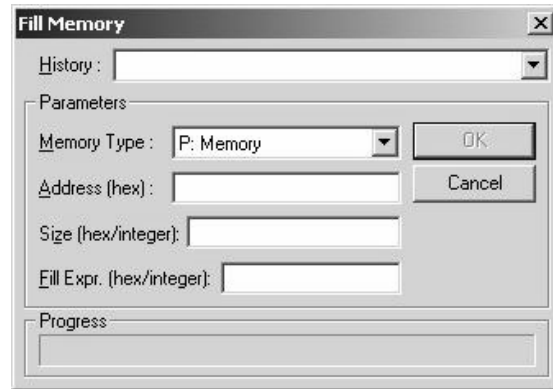
### Browse Button

Clicking the **Browse** button displays OPENFILENAME or SAVEFILENAME, depending on whether you selected the **Load Memory** or **Save Memory** radio button.

## Fill Memory

From the menu bar of the Metrowerks CodeWarrior window, select **Debug > 56800E > Fill Memory** to display the **Fill Memory** dialog box (Figure 9.17).

**Figure 9.17 Fill Memory Dialog Box**



Use this dialog box to fill memory at a specified location and size with user- specified raw memory data. You can associate a key binding with this dialog box for quick access. Press the **Tab** key to cycle through the dialog box display, which lets you quickly make changes without using the mouse.

---

**NOTE** Fill Memory does not support Flash Memory.

---

---

## History Combo Box

The **History** combo box displays a list of recent fill operations. If this is the first time you perform a fill operation, the **History** combo box is empty. If you do more than one fill, then the combo box populates with the memory address of that fill, to a maximum of ten sessions.

If you enter information for an item that already exists in the history list, that item moves up to the top of the list. If you do another fill, then this item is the first one that appears.

---

<b>NOTE</b>	By default, the <b>History</b> combo box displays the most recent settings on subsequent viewings.
-------------	--

---

## Memory Type Combo Box

The memory types that can appear in the **Memory Type** Combo box are:

- P:Memory (Program Memory)
- X:Memory (Data Memory)

## Address Text Field

Specify the address where you want to write the memory. If you want it to be interpreted as hex, prefix it with 0x; otherwise, it is interpreted as decimal.

## Size Text Field

Specify the number of words to write to the target. If you want it to be interpreted as hex, prefix your entry with 0x; otherwise, it is interpreted as decimal.

## Fill Expression Text Field

Fill writes a set of characters to a location specified by the address field on the target, repeatedly copying the characters until the user-supplied fill size has been reached.

**Size** is the total words written, not the number of times to write the string.

## Interpretation of the Fill Expression

The fill string is interpreted differently depending on how it is entered in the Fill String field. Any words prefixed with 0x is interpreted as hex bytes. Thus, 0xBE 0xEF

would actually write 0xBEEF on the target. Optionally, the string could have been set to 0xBEEF and this would do the same thing. Integers are interpreted so that the equivalent signed integer is written to the target.

## ASCII Strings

ASCII strings can be quoted to have literal interpretation of spaces inside the quotes. Otherwise, spaces in the string are ignored. Note that if the ASCII strings are not quoted and they are numbers, it is possible to create illegal numbers. If the number is illegal, an error message is displayed.

## Dialog Box Controls

### OK, Cancel, and Esc

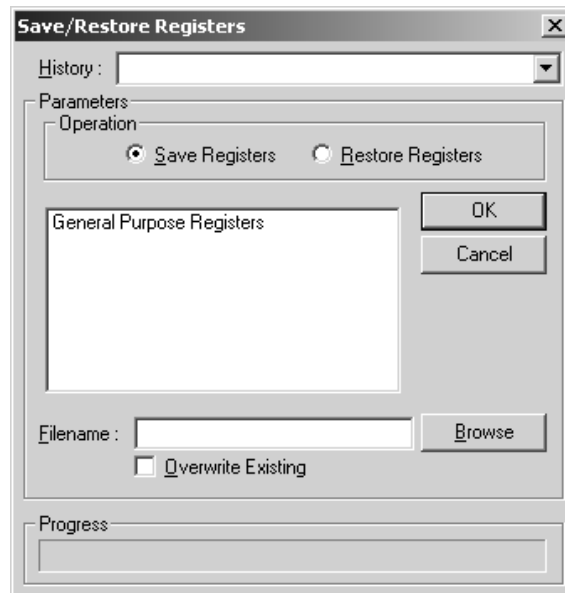
Clicking **OK** writes the memory piece by piece until the target memory is filled in. The **Status** field is updated with the current progress of the operation. When this is in progress, the entire dialog box grays out except the **Cancel** button, so the user cannot change any information. Clicking the **Cancel** button halts the fill operation, and re-enables the controls on the dialog box. Clicking the **Cancel** button again closes the dialog box. Pressing the **Esc** key is same as pressing the **Cancel** button.

## Save/Restore Registers

From the menu bar of the Metrowerks CodeWarrior window, select **Debug > 56800E > Save/Restore Registers** to display the **Save/Restore Registers** dialog box (Figure 9.18).



**Figure 9.18 Save/Restore Registers Dialog Box**



Use this dialog box to save and restore register groups to and from a user-specified file.

## History Combo Box

The **History** combo box displays a list of recent saves and restores. If this is the first time you have saved or restored, the **History** combo box is empty. If you saved or restored before, the combo box remembers your last ten sessions. The most recent session will appear at the top of the list.

## Radio Buttons

The **Save/Restore Registers** dialog box has two radio buttons:

- Save Registers
- Restore Registers

The default is **Save Registers**.

## Register Group List

This list is only available when you have selected **Save Registers**. If you have selected **Restore Registers**, the items in the list are greyed out. Select the register group that you wish to save.

## Dialog Box Controls

### Cancel, Esc, and OK

In Save and Restore operations, all controls are disabled except **Cancel** for the duration of the load or save. The status field is updated with the current progress of the operation. Clicking **Cancel** halts the operation, and re-enables the controls on the dialog box. Clicking **Cancel** again closes the dialog box. Pressing the **Esc** key is same as clicking the **Cancel** button.

With the **Restore Registers** radio button selected, clicking **OK** restores the registers from the specified file and writes it to the registers until the end of the file or the size specified is reached. If the file does not exist, an error message appears.

With the **Save Register** radio button selected, clicking **OK** reads the registers from the target piece by piece and writes it to the specified file. The status field is updated with the current progress of the operation.

### Browse Button

Clicking the **Browse** button displays OPENFILENAME or SAVEFILENAME, depending on whether you selected the **Restore Registers** or **Save Registers** radio button.

## EOnCE Debugger Features

The following EOnCE Debugger features are discussed in this section:

- Set Hardware Breakpoint Panel
- Special Counters
- Trace Buffer
- Set Trigger Panel

---

**NOTE** These features are only available when debugging with a hardware target.

---

For more information on the debugging capabilities of the EOnCE, see the EOnCE chapter of your processor's user manual.

## Set Hardware Breakpoint Panel

The **Set Hardware BreakPoint** panel (Figure 9.19) lets you set a trigger to do one of the following: halt the processor, cause an interrupt, or start or stop trace buffer capture.

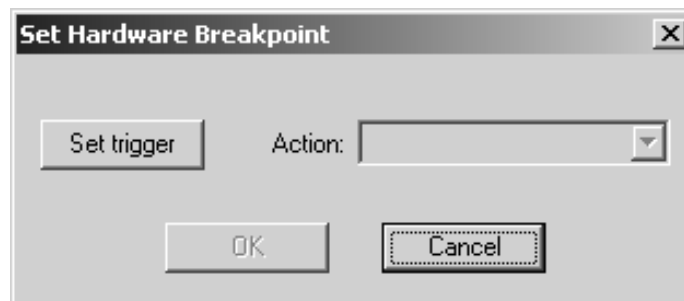
To open this panel:

1. From the menu bar, select DSP56800E > Set Breakpoint Trigger(s).

To clear triggers set with this panel:

1. From the menu bar, select DSP56800E > Clear Triggers.

**Figure 9.19 Set Hardware Breakpoint Panel**



The **Set Hardware BreakPoint** panel options are:

- Set trigger

Select this button to open the **Set Trigger** panel (Figure 9.23). For more information on using this panel, see “Set Trigger Panel” on page 212.

- Action

This pull down list lets you select the resulting action caused by the trigger.

- Halt core  
Stops the processor.

– Interrupt

Causes an interrupt and uses the vector for the EOnCE hardware breakpoint (unit 0).

## Special Counters

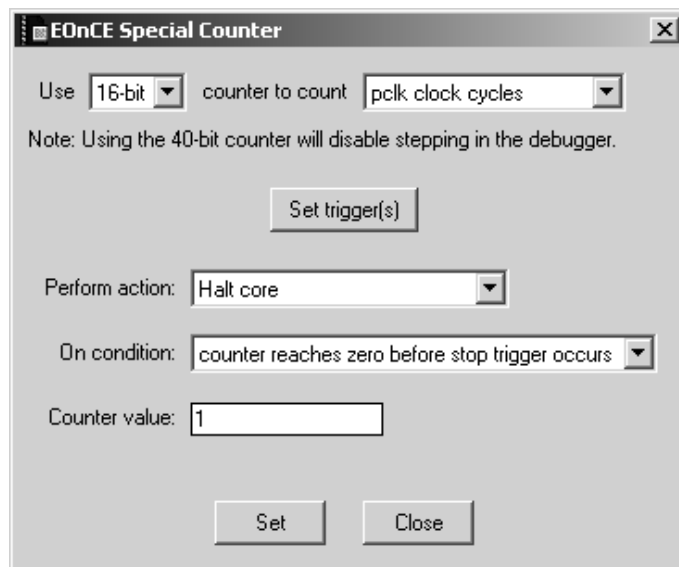
This feature lets you use the special counting function of the EOnCE unit.

To open the EOnCE Special Counter panel (Figure 9.20):

1. From the menu bar, select DSP56800E > Special Counter.

This panel is non-modal and will update itself whenever the processor stops.

**Figure 9.20 EOnCE Special Counter Panel**



The **EOnCE Special Counter** panel options are:

- Counter size

This pull down list gives you the option to use a 16 or 40-bit counter.

---

**NOTE** Using the 40-bit counter will disable stepping in the debugger.

---

- **Counter function**  
This pull down list allows you to choose which counting function to use.
- **Set trigger(s)**  
Pushing this button opens the **Set Trigger** panel. For more information on using this panel, see “Set Trigger Panel” on page 212..
- **Perform action**  
This pull down list lets you select the action that occurs when the correct conditions are met, as set in the **Set Trigger** panel and the **On condition** pull down list.
- **On condition**  
This pull down list lets you set the order in which a trigger and counter reaching zero must occur to perform the action specified in **Perform action**.
- **Counter value**  
This edit box should be preloaded with a non-zero counter value when setting the counter. The counter will proceed backward until a stop condition occurs. The edit box will contain the value of the counter and will be updated whenever the processor stops.

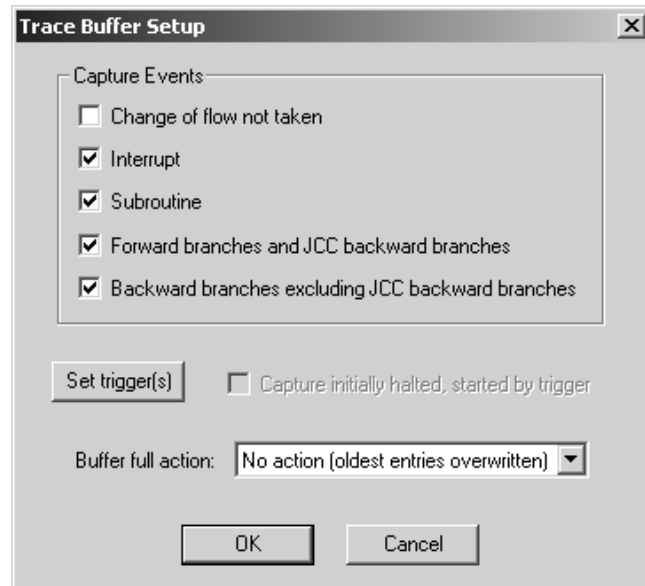
## Trace Buffer

The trace buffer lets you view the target addresses of change-of-flow instructions that the program executes. The trace buffer is configured with the **Trace Buffer Setup** panel (Figure 9.21).

To open this panel:

1. From the IDE menu bar, select DSP56800E > Setup Trace Buffer.

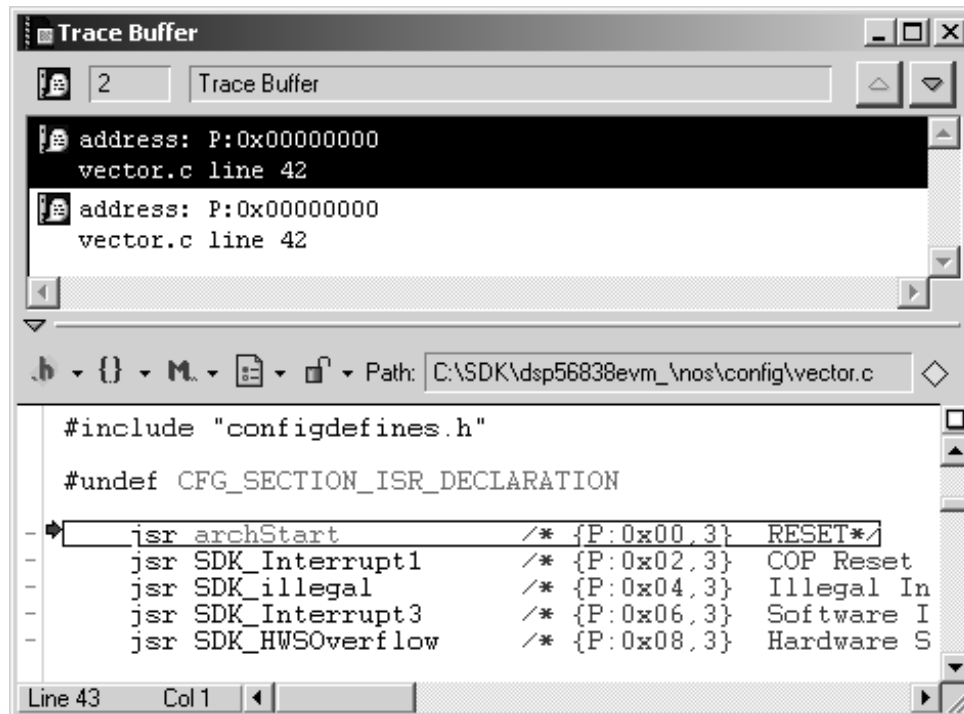
**Figure 9.21 Trace Buffer Setup Panel**



To view the contents of the trace buffer (Figure 9.22):

1. From the IDE menu bar, select DSP56800E > Dump Trace Buffer.

**Figure 9.22 Contents of Trace Buffer**



To clear triggers set with the **Trace Buffer Setup** panel (Figure 9.21):

1. From the menu bar, select DSP56800E > Clear Triggers.

The **Trace Buffer Setup** panel options are:

- **Capture Events**

Select this set of checkboxes to specify which instructions get captured by the trace buffer.

- Change of flow not taken  
Select this checkbox to capture target addresses of conditional branches and jumps that are not taken.
- Interrupt  
Select this checkbox to capture addresses of interrupt vector fetches and target addresses of RTI instructions.
- Subroutine

Select this checkbox to capture target addresses of JSR, BSR, and RTS instructions.

- Forward branches and JCC Backward branches

Select this checkbox to capture target addresses of the following taken instructions:

BCC forward branch

BRSET forward branch

BRCLR forward branch

JCC forward and backward branches

- **Backward branches excluding JCC backward branches**

Select this checkbox to capture target addresses of the following taken instructions:

BCC backward branch

BRSET backward branch

BRCLR backward branch

- Set trigger(s)

Select this button to open the **Set Trigger** panel (Figure 9.23). For more information on using this panel, see “Set Trigger Panel” on page 212.. The resulting trigger halts trace buffer capture.

- **Capture initially halted, started by trigger**

When this option is checked, the trace buffer starts off halted.

- **Buffer full action**

This pull down list lets you select the resulting action caused by the trace buffer filling.

## Set Trigger Panel

The **Set Trigger** panel (Figure 9.23) lets you set triggers for all the EOnCE functions. It can be accessed from the panels used to configure those functions. The options available change depending on the function being configured.



**Figure 9.23 Set Trigger Panel**

The **Set Trigger** panel options are:

- **Primary trigger type**

This pull down list contains the general categories of triggers that can be set.

- **Primary trigger**

This pull down list contains the specific forms of the triggers that can be set. This list changes depending on the selection made in the **Primary trigger type** option. The # symbol contained in some of the triggers' descriptions specifies that the sub-trigger that it precedes must occur the number of times specified in the **Breakpoint counter** option to cause a trigger. The -> symbol specifies that the first sub-trigger must occur, then the second sub-trigger must occur to cause a trigger.

- **Value options**

There are two edit boxes used to specify addresses and data values. The descriptions next to the boxes change according to the selection in **Primary trigger type** and **Primary trigger**. According to these options, only one value may be available.

- **Data compare length**

When the data trigger (address and data) compare trigger is selected, this set of radio buttons becomes available. These options allow you to specify the length of data being compared at that address.

- **Data mask**

When a data compare trigger is selected, this edit box becomes available. This value specifies which bits of the data value are compared.

- **Invert data compare**

When a data compare trigger is selected, this checkbox becomes available. When checked, the comparison result of the data value is inverted (logical NOT).

- **Breakpoint counter**

This edit box specifies the number of times a sub-trigger preceded by a # (see above) must occur to cause a trigger.

- **Advanced trigger**

This pull down list contains options for combining triggers. The types of triggers that can be combined are triggers set in this panel and core events.

- **Core events**

This set of checkboxes specify which core events are allowed to enter the breakpoint logic and cause a trigger.

- **DEBUGEV trigger enabled**

When this checkbox is selected, the **DEBUGEV** instruction causes a core event.

- **Overflow trigger enabled**

When this checkbox is selected, overflow and saturation conditions in the processor cause core events.

- **Use step counter to execute**

When this checkbox is selected, the processor steps through additional instructions after a trigger is signalled. The number of instructions to be stepped is specified in the edit box that is enabled when this checkbox is checked.

## Using the DSP56800E Simulator

The CodeWarrior Development Studio for Freescale 56800/E Hybrid Controllers includes the Freescale DSP56800E Simulator. This software lets you run and debug

code on a simulated DSP56800E architecture without installing any additional hardware.

The simulator simulates the DSP56800E processor, not the peripherals. In order to use the simulator, you must select a connection that uses the simulator as your debugging protocol from the **Remote Debugging** panel.

---

**NOTE** The simulator also enables the DSP56800E menu for retrieving the machine cycle count and machine instruction count when debugging.

---

---

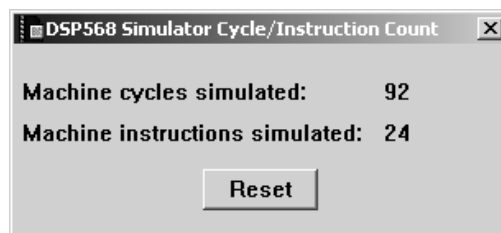
**NOTE** The data memory of the 56800E simulator is read-only from X:0xFF80 to X:0xFFFF.

---

## Cycle/Instruction Count

From the menu bar of the Metrowerks CodeWarrior window, select **56800E > Display Cycle/Instruction count**. The following window appears (Figure 9.24):

**Figure 9.24 Simulator Cycle/Instruction Count**



---

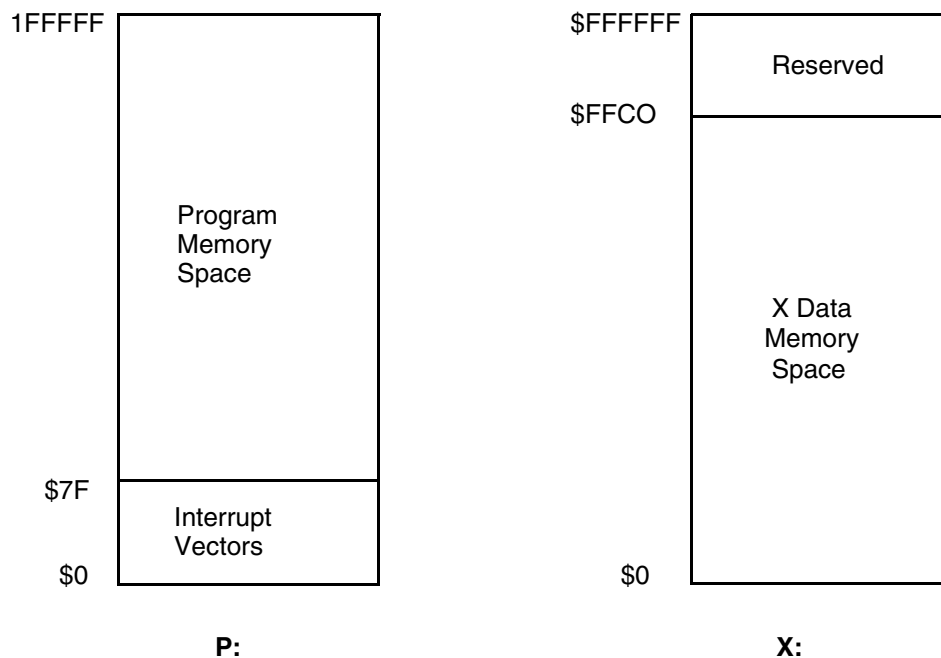
**NOTE** Cycle counting is not accurate while single stepping through source code in the debugger. It is only accurate while running. Thus, the cycle counter is more of a profiling tool than an interactive tool.

---

Press the **Reset** button to zero out the current machine-cycle and machine-instruction readings.

## Memory Map

Figure 9.25 Simulator Memory Map

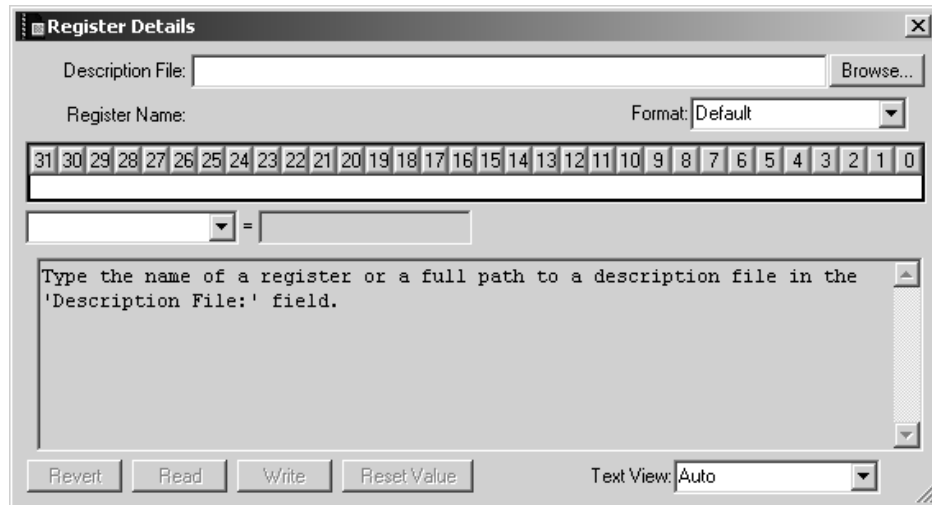


**NOTE** Figure 9.25 is the memory map configuration for the simulator. Therefore, the simulator does not simulate each DSP568xx device's specific memory map, but assumes the memory map of the DSP56824.

## Register Details Window

From the menu bar of the Metrowerks CodeWarrior window, select **View > Register Details** or in the Registers window (Figure 9.9) double-click on the register. The **Register Details** window appears (Figure 9.26).

**Figure 9.26 Register Details Window**



In the **Register Details** window, type the name of the register (e.g., OMR, SR, IPR, etc.) in the **Description File** field. The applicable register and its values appears.

By default, the CodeWarrior IDE looks in the following path when searching for register description files.

```

\CodeWarrior\bin\Plugins\support\Registers
\M56800E\GPR

```

Register description files must end with the `.xml` extension. Alternatively, you can use the **Browse** button to locate the register description files.

Using the **Format** list box in the **Register Details** window, you can change the format in which the CodeWarrior IDE displays the registers.

Using the **Text View** list box in the **Register Details** window, you can change the text information the CodeWarrior IDE displays.

## Loading a .elf File without a Project

You can load and debug a `.elf` file without an associated project. To load a `.elf` file for debugging without an associated project:

1. Launch the CodeWarrior IDE.

## Debugging for DSP56800E Using the Command Window

---

2. Choose **File > Open** and specify the file to load in the standard dialog box that appears.  
Alternatively, you can drag and drop a `.elf` file onto the IDE.
3. You may have to add additional access paths in the Access Path preference panel in order to see all of the source code.
4. Choose **Project > Debug** to begin debugging the application.

---

<b>NOTE</b>	When you debug a <code>.elf</code> file without a project, the IDE sets the <b>Build before running</b> setting on the Build Settings panel of the IDE Preference panels to Never. Consequently, if you open another project to debug after debugging a <code>.elf</code> file, you must change the <b>Build before running</b> setting before you can build the project.
-------------	---

---

The project that the CodeWarrior tools uses to create a new project for the given `.elf` file is `56800E_Default_Project.xml`, which is in the directory located in the path:

CodeWarrior\bin\plugins\support

You can create your own version of this file to use as a default setting when opening a `.elf` file:

1. Create a new project with the default setting you want.
2. Export the project to xml format.
3. Rename the xml format of the project to `56800E_Default_Project.xml` and place it in the support directory.

---

<b>NOTE</b>	Back up or rename the original version of the default xml project before overwriting it with your own customized version.
-------------	---

---

## Using the Command Window

In addition to using the regular CodeWarrior IDE debugger windows, you also can debug using Tcl scripts or the Command Window.

---

For more information on Tcl scripts and the Command Window, please see the *CodeWarrior Development Studio IDE 5.6 Windows® Automation Guide*.

## System-Level Connect

The CodeWarrior DSP56800E debugger lets you connect to a loaded target board and view system registers and memory. A system-level connect does not let you view symbolic information during a connection.

---

<b>NOTE</b>	The following procedure explains how to connect in the context of developing and debugging code on a target board. However, you can select the <b>Debug &gt; Connect</b> command anytime you have a project window open, even if you have not yet downloaded a file to your target board.
-------------	---

---

To perform a system-level connect:

1. Select the Project window for the program you downloaded.
2. From the menu bar, select Debug > Connect.

The debugger connects to the board. You can now examine registers and the contents of memory on the board.

## Debugging in the Flash Memory

The debugger is capable of programming flash memory. The programming occurs at launch, during download. The flash programming option is turned on and the parameters are set in the initialization file. This file is specified in the **Debugger>M56800E Target** preference panel. A list of flash memory commands is given in the next section.

The stationery provides an example of how to specify a default initialization file, how to write a linker command file for flash memory, and how to copy initialized data from ROM to RAM using provided library functions.

---

<b>NOTE</b>	If you use the phase locked loop (PLL) to change the system speed and you are using software or automatic breakpoints, you will need to
-------------	---

---

enable the alternate flash download sequence, as described by the “target\_code\_sets\_hfmclkd” command in the following section.

---

## Flash Memory Commands

The following is a list of flash memory commands that can be included in your initialization file.

For more information on flash memory commands and initialization of the flash, see “M56800E Target (Debugging).”

---

### **set\_hfmclkd <value>**

This command writes the `value` which represents the clock divider for the flash memory to the `hfmclkd` register.

The value for the `set_hfmclkd` command depends on the frequency of the clock. If you are using a supported EVM, this value should not be changed from the value provided in the default initialization file. However, if you are using an unsupported board and the clock frequency is different from that of the supported EVM, a new value must be calculated as described in the user’s manual of the particular processor that you are using.

---

<b>NOTE</b>	The <code>set_hfmclkd</code> , <code>set_hfm_base</code> , and at least one <code>add_hfm_unit</code> command must exist to enable flash programming. All other flash memory commands are optional.
-------------	---

---

---

### **set\_hfm\_base <address>**

This command sets the address of `hfm_base`, which is where the flash control registers are mapped in X: memory.



---

<b>NOTE</b>	The <code>set_hfm_base</code> and <code>add_hfm_unit</code> commands should not be changed for a particular processor. Their values will always be the same.
-------------	--

---

---

## **set\_hfm\_config\_base <address>**

This command sets the address of `hfm_config_base`, which is where the flash security values are written in program flash memory. If this command is present, the debugger used the address to mimic part of the hardware reset behavior by copying the protection values from the configuration field to the appropriate flash control registers.

---

## **add\_hfm\_unit <startAddr> <endAddr> <bank> <numSectors> <page-Size> <progMem> <boot> <interleaved>**

This command adds a flash unit to the list and sets its parameters.

---

<b>NOTE</b>	The <code>set_hfm_base</code> and <code>add_hfm_unit</code> commands should not be changed for a particular processor. Their values will always be the same.
-------------	--

---

---

## **set\_hfm\_erase\_mode units | pages | all**

This command sets the erase mode as `units`, `pages` or `all`. If you set this to `units`, the units that are programmed are mass erased. If set this to `pages`, the pages that are programmed are erased. If you set this to `all`, all units are mass erased including those that have not been programmed. If you omit this command, the erase mode defaults to the unit mode.

---

## set\_hfm\_verify\_erase 1 | 0

If you set this to 1, the debugger verifies that the flash memory has been erased, and alerts you if the erase failed. If this command is omitted, the flash erase is not verified.

---

## set\_hfm\_verify\_program 1 | 0

If you set this to 1, the debugger verifies that the flash has been programmed correctly, and alerts you if the programming failed. If you omit this command, flash programming is not verified.

---

## target\_code\_sets\_hfmcld 1 | 0

If you set this to 1, the debugger uses an alternate launch sequence. First, the flash memory is loaded. Next, the processor is reset to clear the hfmcld register to allow the correct divider to be set for the new system speed (as set by the PLL). Finally, if needed, the RAM is loaded.

When this option is enabled, the hfmcld register needs to be loaded in the startup code. For more details on setting the hfmcld register, see the chapter “Flash Memory” in the *MC56F8300 Peripheral User Manual*. For a demo of the proper use of this feature, see the example code.

## Flash Lock/Unlock

The **Flash Lock** and **Flash Unlock** commands let you control the Flash security state.

The **Flash Lock** command enables the Flash security state. In this state, you can not read the memory or the registers.

The **Flash Unlock** command disables the Flash security. This results in all the Flash memory being erased.

---

<b>NOTE</b>	The <b>Flash Lock</b> and <b>Flash Unlock</b> commands can only be enabled if the debugger session is not running.
-------------	--

---

To use the **Flash Lock** or **Flash Unlock** command:

1. Kill any open debugger sessions.
2. Select a DSP56800E project with a Flash target.

---

<b>NOTE</b>	A Flash target is a target using an initialization file containing Flash commands.
-------------	--

---

3. Select a Flash target.
4. Select either **Debug > 56800E > Flash Lock** or **Debug > 56800E > Flash Unlock** command.

## Notes for Debugging on Hardware

Below are some tips and some things to be aware of when debugging on a hardware target:

- Ensure your Flash data size fits into Flash memory.  
The linker command file specifies where data is written to. There is no bounds checking for Flash programming.
- The standard library I/O function such as `printf` uses large amount of memory and may not fit into flash targets.
- Use the Flash stationery when creating a new project intended for ROM.  
The default stationery contains the Flash configuration file and debugger settings required to use the Flash programmer.
- There is only one hardware breakpoint available, which is shared by IDE breakpoints (when the Breakpoint Mode is set to hardware in the **M56800E Target** panel), watchpoints, and EOnCE triggers. Only one of these may be set at a time.
- When a hardware breakpoint trigger is set to react to an instruction fetch (IDE hardware breakpoint or EOnCE trigger) be aware that the hardware will react to the fetch whether or not the fetched instruction is executed. For example, if a hardware breakpoint is set just after a loop, the processor will stop with the execution point inside the loop. This is because the target instruction will be fetched while the program is in the loop due to the large pipeline. A branch will occur to facilitate the loop; however, the processor will stop because the target instruction has already been fetched.

## Debugging for DSP56800E

*Notes for Debugging on Hardware*

---

- The M56800E cannot single step over certain two and three-word uninterrupted sequences. However, the debugger compensates using software breakpoints and the trace buffer to allow single stepping in these situations. But, if these techniques cannot be used (e.g., debugging in ROM or the trace buffer in use) single stepping over these sequences results in the processor executing each instruction in the sequence before stopping. The execution will be correct. Just be aware of this "slide" in these situations.

# Profiler

---

The profiler is a run-time feature that collects information about your program. It records the minimum, maximum, and total number of clock cycles spent in each function. The profiler allows you to evaluate your code and determine which functions require optimization.

When profiling is enabled, the compiler adds code to call the entry functions in the profiler library. These profiler library functions do all of the data collection. The profiler library, with the help of the debugger create a binary output file, which is opened and displayed by the CodeWarrior IDE.

---

<b>NOTE</b>	For more information on the profiler library and its usage, see the <i>CodeWarrior Development Studio IDE 5.5 User's Guide Profiler Supplement</i> .
-------------	--

---

To enable your project for profiling:

1. Add the following path to your list of user paths in the Access Paths settings panel:  

```
{Compiler}M56800x Support\profiler
```
2. Add the following line to the file that contains the function main():  

```
#include "Profiler.h"
```
3. Add the profiler library file to your project. Select the library that matches your target from this path:

```
{CodeWarrior path}M56800x Support\profiler\lib
```

## Profiler

---

4. Add the following function calls to main():

```
ProfilerInit()  
ProfilerClear()  
ProfilerSetStatus()  
ProfilerDump()  
ProfilerTerm()
```

For more details of these functions, see the *CodeWarrior Development Studio IDE 5.5 User's Guide Profiler Supplement*.

5. It may be necessary to increase the heap size to accommodate the profiler data collection. This can be set in the linker command file by changing the value of `__heap_size`.
6. Enable profiling by setting the **Generate code for profiling** option in the **M56800E Processor** settings panel or by using the `profile on | off` pragma to select individual functions to profile.

---

<b>NOTE</b>	For a profiler example, see the profiler example in this path: {CodeWarrior path}\CodeWarrior_Examples\SimpleProfiler
-------------	--

---

## Inline Assembly Language and Intrinsic Functions

---

The CodeWarrior™ compiler supports inline assembly language and intrinsic functions. This chapter explains the IDE implementation of Freescale assembly language, with regard to DSP56800E development. It also explains the relevant intrinsic functions.

This chapter contains these sections:

- Inline Assembly Language
- Intrinsic Functions

### Inline Assembly Language

This section explains how to use inline assembly language. It contains these sections:

- Inline Assembly Overview
- Assembly Language Quick Guide
- Calling Assembly Language Functions from C Code
- Calling Functions from Assembly Language

## Inline Assembly Overview

To specify assembly-language interpretation for a block of code in your file, use the `asm` keyword and standard DSP56800E instruction mnemonics.

---

<b>NOTE</b>	To make sure that the C compiler recognizes the <code>asm</code> keyword, you must clear the <b>ANSI Keywords Only</b> checkbox of the <b>C/C++ Language (C Only)</b> settings panel.
	Differences in calling conventions mean that you cannot re-use DSP56800 assembly code in the DSP56800E compiler.

---

Listing 11.1 shows how to use the `asm` keyword with braces, to specify that an *entire function* is in assembly language.

### Listing 11.1 Function-Level Syntax

---

```
asm <function header>
{
    <assembly instructions>
}
```

---

The function header can be any valid C function header; the local declarations are any valid C local declarations.

Listing 11.2 shows how to use the `asm` keyword with braces, to specify that a block of statements or a single statement is in assembly language.

### Listing 11.2 Statement-Level Syntax

---

```
asm { inline assembly statement
      inline assembly statement
      ...
}

asm {inline assembly statement}
```

---

The inline assembly statement is any valid assembly-language statement.

Listing 11.3 shows how to use the `asm` keyword with parentheses, to specify that a single statement is in assembly language. Note that a semicolon must follow the close parenthesis.



---

## Listing 11.3 Alternate Single-Statement Syntax

---

```
asm (inline assembly statement);
```

---

---

<b>NOTE</b>	If you apply the <code>asm</code> keyword to one statement or a block of statements <i>within a function</i> , you must <i>not</i> define local variables within any of the inline-assembly statements.
-------------	---

---

## Assembly Language Quick Guide

Keep these rules in mind as you write assembly language functions:

1. Each statement must be a *label* or a *function*.
2. A *label* can be any identifier not already declared as a local variable.
3. All *labels* must follow the syntax:

`[LocalLabel:]`

Listing 11.4 illustrates the use of labels.

---

## Listing 11.4 Labels in M56800E Assembly

---

```
x1:  add  x0,y1,a
x2:
    add  x0,y1,a
x3   add  x0,y1,a  //ERROR, MISSING COLON
```

---

4. All *instructions* must follow the syntax:  
`( (instruction) [operands] )`
5. Each statement must end with a new line

## Inline Assembly Language and Intrinsics

### Inline Assembly Language

---

6. Assembly language directives, instructions, and registers are not case-sensitive. The following two statements are the same:

```
add    x0,y0
ADD    X0,Y0
```

7. Comments must have the form of C or C++ comments; they must not begin with the ; or # characters. Listing 11.5 shows the valid syntax for comments.

#### Listing 11.5 Valid Comment Syntax

---

```
move.w    x:(r3),y0    # ERROR
add.w     x0,y0         // OK
move.w     r2,x:(sp)    ; ERROR
adda      r0,r1,n       /* OK */
```

---

8. To optimize a block of inline assembly source code, use the inline assembly directive `.optimize_iasm` on before the code block. Then use the directive `.optimize_iasm off` at the end of the block. (Omitting `.optimize_iasm off` means that optimizations continue to the end of the function.)

## Calling Assembly Language Functions from C Code

You can call assembly language functions from C just as you would call any standard C function, using standard C syntax.

### Calling Inline Assembly Language Functions

Listing 11.6 demonstrates how to create an inline assembly language function in a C source file. This example adds two 16-bit integers and returns the result.

Notice that you are passing two 16-bit addresses to the `add_int` function. You pick up those addresses in R2 and R3, passing the sum back in Y0.

#### Listing 11.6 Sample Code - Creating an Inline Assembly Language Function

---

```
asm int add_int( int * i, int * j )
{
    move.w    x:(r2),y0
    move.w    x:(r3),x0
```

---

---

```
add    x0,y0
// int result returned in y0
rts
}
```

---

Listing 11.7 shows the C calling statement for this inline-assembly-language function.

## Listing 11.7 Sample Code - Calling an Inline Assembly Language Function

---

```
int x = 4, y = 2;

y = add_int( &x, &y ); /* Returns 6 */
```

---

## Calling Pure Assembly Language Functions

If you want C code to call assembly language files, you must specify a `SECTION` mapping for your code, for appropriate linking. You must also specify a memory space location. Usually, this means that the `ORG` directive specifies code to program memory (P) space.

In the definition of an assembly language function, the `GLOBAL` directive must specify the current-section symbols that need to be accessible by other sections.

Listing 11.8 is an example of a complete assembly language function. This function writes two 16-bit integers to program memory. A separate function is required for writing to P: memory, because C pointer variables allow access only to X: data memory.

The first parameter is a short value and the second parameter is the 16-bit address.

## Listing 11.8 Sample Code - Creating an Assembly Language Function

---

```
                                ;"my_asm.asm"
SECTION user                    ;map to user defined section in CODE
ORG P:                          ;put the following program in P
                                ;memory

GLOBAL Fpmemwrite               ;This symbol is defined within the
                                ;current section and should be
                                ;accessible by all sections
Fpmemwrite:
    MOVE    Y1,R0                ;Set up pointer to address
    NOP                      ;Pipeline delay for R0
    MOVE    Y0,P:(R0)+           ;Write 16-bit value to address
```

---

## Inline Assembly Language and Intrinsic *Inline Assembly Language*

---

```
                                ;pointed to by R0 in P: memory and
                                ;post-increment R0
rts                            ;return to calling function

ENDSEC                         ;End of section
END                            ;End of source program
```

---

Listing 11.9 shows the C calling statement for this assembly language function.

### Listing 11.9 Sample Code - Calling an Assembly Language Function from C

---

```
void pmemwrite( short, short );/* Write a value into P: memory */

void main( void )
{
    // ...other code

    // Write the value given in the first parameter to the address
    // of the second parameter in P: memory
    pmemwrite( (short)0xE9C8, (short)0x0010 );

    // other code...
}
```

---

## Calling Functions from Assembly Language

Assembly language programs can call functions written in either C or assembly language.

- From within assembly language instructions, you can call C functions. For example, if the C function definition is:

```
void foot( void ) {
    /* Do something */
}
```

Your assembly language calling statement is:

```
jsr  Ffoot
```

- From within assembly language instructions, you can call assembly language functions. For example, if `pmemwrite` is an assembly language function, the assembly language calling statement is:

```
jsr  Fpmemwrite
```

## Intrinsic Functions

This section explains CodeWarrior intrinsic functions. It consists of these sections:

- Implementation
- Fractional Arithmetic
- Intrinsic Functions for Math Support
- Modulo Addressing Intrinsic Functions

### Implementation

The CodeWarrior IDE for DSP56800E has intrinsic functions to generate inline-assembly-language instructions. These intrinsic functions are a CodeWarrior extension to ANSI C.

Use intrinsic functions to target specific processor instructions. For example:

- Intrinsic functions let you pass in data for specific optimized computations. For example, ANSI C data-representation rules may make certain calculations inefficient, forcing the program to jump to runtime math routines. Such calculations would be coded more efficiently as assembly language instructions and intrinsic functions.
- Intrinsic functions can control small tasks, such as enabling saturation. One method is using inline assembly language syntax, specifying the operation in an asm block, every time that the operation is required. But intrinsic functions let you merely set the appropriate bit of the operating mode register.

The IDE implements intrinsic functions as inline C functions in file `intrinsics_56800E.h`, in the MSL directory tree. These inline functions contain mostly inline assembly language code. An example is the `abs_s` intrinsic, defined as:

#### Listing 11.10 Example Code - Definition of Intrinsic Function: `abs_s`

```
#define    abs_s(a)  __abs_s(a)
          /* ABS_S */

inline Word16 __abs_s(register Word16 svar1)
{
/*
 *   Defn: Absolute value of a 16-bit integer or fractional value
 *         returning a 16-bit result.
 *         Returns $7fff for an input of $8000
 *
 *   DSP56800E instruction syntax:  abs FFF
 *         Allowed src regs:  FFF
 */
}
```

## Inline Assembly Language and Intrinsics Intrinsic Functions

---

```

*           Allowed dst regs:  (same)
*
*   Assumptions: OMR's SA bit was set to 1 at least 3 cycles
*   before this code.
*/
asm(abs svar1);
return svar1;
}

```

---

## Fractional Arithmetic

Many of the intrinsic functions use fractional arithmetic with *implied fractional values*. An implied fractional value is a symbol declared as an integer type, but calculated as a fractional type. Data in a memory location or register can be interpreted as fractional or integer, depending on program needs.

All intrinsic functions that generate multiply or divide instructions perform fractional arithmetic on implied fractional values. (These intrinsic functions are DIV, MPY, MAC, MPYR, and MACR) The relationship between a 16-bit integer and a fractional value is:

$$\text{Fractional Value} = \text{Integer Value} / (2^{15})$$

The relationship between a 32-bit integer and a fractional value is similar:

$$\text{Fractional Value} = \text{Long Integer Value} / (2^{31})$$

Table 11.1 shows how 16- and 32-bit values can be interpreted as either fractional or integer values.

**Table 11.1 Interpretation of 16- and 32-bit Values**

Type	Hex	Integer Value	Fixed-point Value
short int	0x2000	8192	0.25
short int	0xE000	-8192	-0.25
long int	0x20000000	536870912	0.25
long int	0xE0000000	-536870912	-0.25

---

<b>NOTE</b>	Intrinsic functions use these macros:
	Word16. — A macro for signed short.
	Word32. — A macro for signed long.

---

## Intrinsic Functions for Math Support

Table 11.2 lists the math intrinsic functions. See section “Modulo Addressing Intrinsic Functions” on page 270. for explanations of the remaining intrinsic functions.

For the latest information about intrinsic functions, refer to file  
`intrinsics_56800E.h`.

**Table 11.2 Intrinsic Functions for DSP56800E**

Category	Function	Category (cont.)	Function (cont.)
Absolute/Negate	abs_s	Multiplication/MAC	mac_r
	negate		msu_r
	L_abs		mult
	L_negate		mult_r
Addition/ Subtraction	add		L_mac
	sub		L_msu
	L_add		L_mult
	L_sub		L_mult_ls
Control	stop	Normalization	ffs_s
	wait		norm_s
	turn_off_conv_rndg		ffs_l
	turn_off_sat		norm_l
	turn_on_conv_rndg	Rounding	round
	turn_on_sat	Shifting	shl
Deposit/Extract	extract_h		shlftNs
	extract_l		shlfts
	L_deposit_h		shr
	L_deposit_l		shr_r
Division	div_s		shrtNs
	div_s4q		L_shl
	div_ls		L_shlftNs
	div_ls4q		L_shlfts
			L_shr
			L_shr_r
			L_shrtNs



## Absolute/Negate

The intrinsic functions of the absolute-value/negate group are:

- `abs_s`
- `negate`
- `L_abs`
- `L_negate`

---

### **`abs_s`**

Absolute value of a 16-bit integer or fractional value returning a 16-bit result. Returns 0x7FFF for an input of 0x8000.

#### **Assumptions**

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

#### **Prototype**

```
Word16 abs_s(Word16 svar1)
```

#### **Example**

```
int result, s1 = 0xE000; /* - 0.25 */
result = abs_s(s1);
// Expected value of result: 0x2000 = 0.25
```

---

### **`negate`**

Negates a 16-bit integer or fractional value returning a 16-bit result. Returns 0x7FFF for an input of 0x8000.

#### **Assumptions**

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

## Inline Assembly Language and Intrinsics *Intrinsic Functions*

---

### Prototype

```
Word16 negate(Word16 svar1)
```

### Example

```
int result, s1 = 0xE000; /* - 0.25 */
result = negate(s1);
// Expected value of result: 0x2000 = 0.25
```

---

## L\_abs

Absolute value of a 32-bit integer or fractional value returning a 32-bit result. Returns 0x7FFFFFFF for an input of 0x80000000.

### Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

### Prototype

```
Word32 L_abs(Word32 lvar1)
```

### Example

```
long result, l = 0xE0000000; /* - 0.25 */
result = L_abs(s1);
// Expected value of result: 0x20000000 = 0.25
```

---

## L\_negate

Negates a 32-bit integer or fractional value returning a 32-bit result. Returns 0x7FFFFFFF for an input of 0x80000000.

### Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

---

## Prototype

```
Word32 L_negate(Word32 lvar1)
```

## Example

```
long result, l = 0xE0000000; /* - 0.25 */
result = L_negate(s1);
// Expected value of result: 0x20000000 = 0.25
```

## Addition/Subtraction

The intrinsic functions of the addition/subtraction group are:

- add
- sub
- L\_add
- L\_sub

---

## add

Addition of two 16-bit integer or fractional values, returning a 16-bit result.

## Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

## Prototype

```
Word16 add(Word16 src_dst, Word16 src2)
```

## Inline Assembly Language and Intrinsic Functions

---

### Example

```
short s1 = 0x4000; /* 0.5 */
short s2 = 0x2000; /* 0.25 */
short result;

result = add(s1,s2);
// Expected value of result: 0x6000 = 0.75
```

---

## sub

Subtraction of two 16-bit integer or fractional values, returning a 16-bit result.

### Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

### Prototype

```
Word16 sub(Word16 src_dst, Word16 src2)
```

### Example

```
short s1 = 0x4000; /* 0.5 */
short s2 = 0xE000; /* -0.25 */
short result;

result = sub(s1,s2);
// Expected value of result: 0x6000 = 0.75
```

---

## L\_add

Addition of two 32-bit integer or fractional values, returning a 32-bit result.

### Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

### Prototype

```
Word32 L_add(Word32 src_dst, Word32 src2)
```

### Example

```
long la = 0x40000000; /* 0.5 */
long lb = 0x20000000; /* 0.25 */
long result;

result = L_add(la, lb);
// Expected value of result: 0x60000000 = 0.75
```

---

## L\_sub

Subtraction of two 32-bit integer or fractional values, returning a 32-bit result.

### Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

### Prototype

```
Word32 L_sub(Word32 src_dst, Word32 src2)
```

## Inline Assembly Language and Intrinsics

### *Intrinsic Functions*

---

#### Example

```
long la = 0x40000000; /* 0.5 */
long lb = 0xE0000000; /* -0.25 */
long result;

result = L_sub(la,lb);
// Expected value of result: 0x60000000 = 0.75
```

## Control

The intrinsic functions of the control group are:

- stop
- wait
- turn\_off\_conv\_rndg
- turn\_off\_sat
- turn\_on\_conv\_rndg
- turn\_on\_sat

---

## stop

Generates a STOP instruction which places the processor in the low power STOP mode.

### Prototype

```
void stop(void)
```

### Usage

```
stop();
```

---

## wait

Generates a WAIT instruction which places the processor in the low power WAIT mode.

### Prototype

```
void wait(void)
```

### Usage

```
wait();
```

---

## turn\_off\_conv\_rndg

Generates a sequence for disabling convergent rounding by setting the R bit in the OMR register and waiting for the enabling to take effect.

---

<b>NOTE</b>	If convergent rounding is disabled, the assembler performs 2's complement rounding.
-------------	---

---

### Prototype

```
void turn_off_conv_rndg(void)
```

### Usage

```
turn_off_conv_rndg();
```

---

## turn\_off\_sat

Generates a sequence for disabling automatic saturation in the MAC Output Limiter by clearing the SA bit in the OMR register and waiting for the disabling to take effect.

## Inline Assembly Language and Intrinsic Functions

---

### Prototype

```
void turn_off_sat(void)
```

### Usage

```
turn_off_sat();
```

---

## turn\_on\_conv\_rndg

Generates a sequence for enabling convergent rounding by clearing the R bit in the OMR register and waiting for the enabling to take effect.

### Prototype

```
void turn_on_conv_rndg(void)
```

### Usage

```
turn_on_conv_rndg();
```

---

## turn\_on\_sat

Generates a sequence for enabling automatic saturation in the MAC Output Limiter by setting the SA bit in the OMR register and waiting for the enabling to take effect.

### Prototype

```
void turn_on_sat(void)
```

### Usage

```
turn_on_sat();
```

---



## Deposit/Extract

The intrinsic functions of the deposit/extract group are:

- `extract_h`
- `extract_l`
- `L_deposit_h`
- `L_deposit_l`

---

### `extract_h`

Extracts the 16 MSBs of a 32-bit integer or fractional value. Returns a 16-bit value. Does not perform saturation. When an accumulator is the destination, zeroes out the LSP portion. Corresponds to *truncation* when applied to fractional values.

#### Prototype

```
Word16 extract_h(Word32 lsrc)
```

#### Example

```
long l = 0x87654321;
short result;

result = extract_h(l);
// Expected value of result: 0x8765
```

---

### `extract_l`

Extracts the 16 LSBs of a 32-bit integer or fractional value. Returns a 16-bit value. Does not perform saturation. When an accumulator is the destination, zeroes out the LSP portion.

#### Prototype

```
Word16 extract_l(Word32 lsrc)
```

## Inline Assembly Language and Intrinsic Functions

---

### Example

```
long l = 0x87654321;
short result;

result = extract_l(l);
// Expected value of result: 0x4321
```

---

## L\_deposit\_h

Deposits the 16-bit integer or fractional value into the upper 16 bits of a 32-bit value, and zeroes out the lower 16 bits of a 32-bit value.

### Prototype

```
Word32 L_deposit_h(Word16 ssrc)
```

### Example

```
short s1 = 0x3FFF;
long result;

result = L_deposit_h(s1);
// Expected value of result: 0x3fff0000
```

---

## L\_deposit\_l

Deposits the 16-bit integer or fractional value into the lower 16 bits of a 32-bit value, and sign extends the upper 16 bits of a 32-bit value.

### Prototype

```
Word32 L_deposit_l(Word16 ssrc)
```

## Example

```
short s1 = 0x7FFF;  
long result;  
  
result = L_deposit_l(s1);  
// Expected value of result: 0x00007FFF
```

## Division

The intrinsic functions of the division group are:

- `div_s`
- `div_s4q`
- `div_ls`
- `div_ls4q`

---

## `div_s`

Single quadrant division, that is, both operands are of positive 16-bit fractional values, returning a 16-bit result. If both operands are equal, returns 0x7FFF (occurs naturally).

---

<b>NOTE</b>	Does not check for division overflow or division by zero.
-------------	---

---

## Prototype

```
Word16 div_s(Word16 s_numerator, Word16 s_denominator)
```

## Inline Assembly Language and Intrinsics Intrinsic Functions

---

### Example

```
short s1=0x2000; /* 0.25 */
short s2=0x4000; /* 0.5 */
short result;

result = div_s(s1,s2);
// Expected value of result: 0.25/0.5 = 0.5 = 0x4000
```

---

## div\_s4q

Four quadrant division of two 16-bit fractional values, returning a 16-bit result.

---

<b>NOTE</b>	Does not check for division overflow or division by zero.
-------------	---

---

### Prototype

```
Word16 div_s4q(Word16 s_numerator, Word16 s_denominator)
```

### Example

```
short s1=0xE000; /* -0.25 */
short s2=0xC000; /* -0.5 */
short result;

result = div_s4q(s1,s2);
// Expected value of result: -0.25/-0.5 = 0.5 = 0x4000
```

---

## div\_ls

Single quadrant division, that is, both operands are positive two 16-bit fractional values, returning a 16-bit result. If both operands are equal, returns 0x7FFF (occurs naturally).

---

<b>NOTE</b>	Does not check for division overflow or division by zero.
-------------	---

---

## Prototype

```
Word16 div_ls(Word32 l_numerator, Word16 s_denominator)
```

## Example

```
long l = 0x20000000; /* 0.25 */
short s2 = 0x4000; /* 0.5 */
short result;

result = div_ls(l, s2);
// Expected value of result: 0.25/0.5 = 0.5 = 0x4000
```

---

## div\_ls4q

Four quadrant division of a 32-bit fractional dividend and a 16-bit fractional divisor, returning a 16-bit result.

---

<b>NOTE</b>	Does not check for division overflow or division by zero.
-------------	---

---

## Prototype

```
Word16 div_ls4q(Word32 l_numerator, Word16 s_denominator)
```

## Example

```
long l = 0xE0000000; /* -0.25 */
short s2 = 0xC000; /* -0.5 */
short result;

result = div_ls4q(s1, s2);
// Expected value of result: -0.25/-0.5 = 0.5 = 0x4000
```

## Multiplication/MAC

The intrinsic functions of the multiplication/MAC group are:

- `mac_r`
- `msu_r`
- `mult`
- `mult_r`
- `L_mac`
- `L_msu`
- `L_mult`
- `L_mult_ls`

---

### **mac\_r**

Multiply two 16-bit fractional values and add to 32-bit fractional value. Round into a 16-bit result, saturating if necessary. When an accumulator is the destination, zeroes out the LSP portion.

#### **Assumptions**

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

OMR's R bit was set to 1 at least 3 cycles before this code, that is, 2's complement rounding, not convergent rounding.

#### **Prototype**

```
Word16 mac_r(Word32 laccum, Word16 sinp1, Word16 sinp2)
```

## Example

```
short s1 = 0xC000; /* - 0.5 */
short s2 = 0x4000; /*  0.5 */
short result;
long Acc = 0x0000FFFF;

result = mac_r(Acc,s1,s2);
// Expected value of result: 0xE001
```

---

## msu\_r

Multiply two 16-bit fractional values and subtract this product from a 32-bit fractional value. Round into a 16-bit result, saturating if necessary. When an accumulator is the destination, zeroes out the LSP portion.

## Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

OMR's R bit was set to 1 at least 3 cycles before this code, that is, 2's complement rounding, not convergent rounding.

## Prototype

```
Word16 msu_r(Word32 laccum, Word16 sinp1, Word16 sinp2)
```

## Example

```
short s1 = 0xC000; /* - 0.5 */
short s2 = 0x4000; /*  0.5 */
short result;
long Acc = 0x20000000;

result = msu_r(Acc,s1,s2);
// Expected value of result: 0x4000
```

## mult

Multiply two 16-bit fractional values and truncate into a 16-bit fractional result. Saturates only for the case of 0x8000 x 0x8000. When an accumulator is the destination, zeroes out the LSP portion.

### Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

### Prototype

```
Word16 mult(Word16 sinp1, Word16 sinp2)
```

### Example

```
short s1 = 0x2000; /* 0.25 */
short s2 = 0x2000; /* 0.25 */
short result;

result = mult(s1,s2);
// Expected value of result: 0.625 = 0x0800
```

---

## mult\_r

Multiply two 16-bit fractional values, round into a 16-bit fractional result. Saturates only for the case of 0x8000 x 0x8000. When an accumulator is the destination, zeroes out the LSP portion.

### Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

OMR's R bit was set to 1 at least 3 cycles before this code, that is, 2's complement rounding, not convergent rounding.



## Prototype

```
Word16 mult_r(Word16 sinp1, Word16 sinp2)
```

## Example

```
short s1 = 0x2000; /* 0.25 */
short s2 = 0x2000; /* 0.25 */
short result;

result = mult_r(s1,s2);
// Expected value of result: 0.0625 = 0x0800
```

---

## L\_mac

Multiply two 16-bit fractional values and add to 32-bit fractional value, generating a 32-bit result, saturating if necessary.

## Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

## Prototype

```
Word32 L_mac(Word32 laccum, Word16 sinp1, Word16 sinp2)
```

## Example

```
short s1 = 0xC000; /* - 0.5 */
short s2 = 0x4000; /* 0.5 */
long result, Acc = 0x20000000; /* 0.25 */

result = L_mac(Acc,s1,s2);
// Expected value of result: 0
```

### L\_msu

Multiply two 16-bit fractional values and subtract this product from a 32-bit fractional value, saturating if necessary. Generates a 32-bit result.

#### Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

#### Prototype

```
Word32 L_msu(Word32 laccum, Word16 sinp1, Word16 sinp2)
```

#### Example

```
short s1 = 0xC000; /* - 0.5 */
short s2 = 0xC000; /* - 0.5 */
long result, Acc = 0;

result = L_msu(Acc, s1, s2);
// Expected value of result: 0.25
```

---

### L\_mult

Multiply two 16-bit fractional values generating a signed 32-bit fractional result. Saturates only for the case of 0x8000 x 0x8000.

#### Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

#### Prototype

```
Word32 L_mult(Word16 sinp1, Word16 sinp2)
```

## Example

```
short s1 = 0x2000; /* 0.25 */
short s2 = 0x2000; /* 0.25 */
long result;

result = L_mult(s1,s2);
// Expected value of result: 0.0625 = 0x08000000
```

---

## L\_mult\_ls

Multiply one 32-bit and one-16-bit fractional value, generating a signed 32-bit fractional result. Saturates only for the case of 0x80000000 x 0x8000.

## Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

## Prototype

```
Word32 L_mult_ls(Word32 linp1, Word16 sinp2)
```

## Example

```
long l1 = 0x20000000; /* 0.25 */
short s2 = 0x2000; /* 0.25 */
long result;

result = L_mult(l1,s2);
// Expected value of result: 0.625 = 0x08000000
```

## Normalization

The intrinsic functions of the normalization group are:

- `ffs_s`
- `norm_s`
- `ffs_l`
- `norm_l`

---

### `ffs_s`

Computes the number of left shifts required to normalize a 16-bit value, returning a 16-bit result (finds 1st sign bit). Returns a shift count of 31 for an input of 0x0000.

---

<b>NOTE</b>	Does not actually normalize the value! Also see the intrinsic <code>norm_s</code> which handles the case where the input == 0x0000 differently.
-------------	---

---

### Prototype

```
Word16 ffs_s(Word16 ssrc)
```

### Example

```
short s1 = 0x2000; /* .25 */
short result;

result = ffs_s(s1);
// Expected value of result: 1
```

---

### `norm_s`

Computes the number of left shifts required to normalize a 16-bit value, returning a 16-bit result. Returns a shift count of 0 for an input of 0x0000.

---

<b>NOTE</b>	Does not actually normalize the value! This operation is <i>not</i> optimal on the DSP56800E because of the case of returning 0 for an input of 0x0000. See the intrinsic <code>ffs_s</code> which is more optimal but generates a different value for the case where the input == 0x0000.
-------------	--

---

## Prototype

```
Word16 norm_s(Word16 ssrc)
```

## Example

```
short s1 = 0x2000; /* .25 */
short result;

result = norm_s(s1);
// Expected value of result: 1
```

---

## ffs\_l

Computes the number of left shifts required to normalize a 32-bit value, returning a 16-bit result (finds 1st sign bit). Returns a shift count of 31 for an input of 0x00000000.

---

<b>NOTE</b>	Does not actually normalize the value! Also, see the intrinsic <code>norm_l</code> which handles the case where the input == 0x00000000 differently.
-------------	--

---

## Prototype

```
Word16 ffs_l(Word32 lsrc)
```

## Inline Assembly Language and Intrinsics

### *Intrinsic Functions*

---

#### Example

```
long ll = 0x20000000; /* .25 */
short result;

result = ffs_l(ll);
// Expected value of result: 1
```

---

## norm\_l

Computes the number of left shifts required to normalize a 32-bit value, returning a 16-bit result. Returns a shift count of 0 for an input of 0x00000000.

---

<b>NOTE</b>	Does not actually normalize the value! This operation is <i>not</i> optimal on the DSP56800E because of the case of returning 0 for an input of 0x00000000. See the intrinsic ffs_l which is more optimal but generates a different value for the case where the input == 0x00000000.
-------------	---

---

#### Prototype

```
Word16 norm_l(Word32 lsrc)
```

#### Example

```
long ll = 0x20000000; /* .25 */
short result;

result = norm_l(ll);
// Expected value of result: 1
```

## Rounding

The intrinsic function of the rounding group is:

- round

---

### round

Rounds a 32-bit fractional value into a 16-bit result. When an accumulator is the destination, zeroes out the LSP portion.

#### Assumptions

OMR's R bit was set to 1 at least 3 cycles before this code, that is, 2's complement rounding, not convergent rounding.

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

#### Prototype

```
Word16 round(Word32 lvar1)
```

#### Example

```
long l = 0x12348002; /*if low 16 bits = 0xFFFF > 0x8000 then  
add 1 */  
short result;  
  
result = round(l);  
// Expected value of result: 0x1235
```

## Shifting

The intrinsic functions of the shifting group are:

- shl
- shlftNs
- shlfts
- shr
- shr\_r
- shrtNs
- L\_shl
- L\_shlftNs
- L\_shlfts
- L\_shr
- L\_shr\_r
- L\_shrtNs

---

### shl

Arithmetic shift of 16-bit value by a specified shift amount. If the shift count is positive, a left shift is performed. Otherwise, a right shift is performed. Saturation may occur during a left shift. When an accumulator is the destination, zeroes out the LSP portion.

---

<b>NOTE</b>	This operation is not optimal on the DSP56800E because of the saturation requirements and the bidirectional capability. See the intrinsic shlftNs or shlfts which are more optimal.
-------------	---

---

### Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

### Prototype

```
Word16 shl(Word16 sval2shft, Word16 s_shftamount)
```



---

## Example

```
short result;  
short s1 = 0x1234;  
short s2 = 1;  
  
result = shl(s1,s2);  
// Expected value of result: 0x2468
```

---

## shlftNs

Arithmetic shift of 16-bit value by a specified shift amount. If the shift count is positive, a left shift is performed. Otherwise, a right shift is performed. Saturation does not occur during a left shift. When an accumulator is the destination, zeroes out the LSP portion.

---

<b>NOTE</b>	Ignores upper N-5 bits of s_shftamount except the sign bit (MSB). If s_shftamount is positive and the value in the lower 5 bits of s_shftamount is greater than 15, the result is 0. If s_shftamount is negative and the absolute value in the lower 5 bits of s_shftamount is greater than 15, the result is 0 if sval2shft is positive, and 0xFFFF if sval2shft is negative.
-------------	--

---

## Prototype

```
Word16 shlftNs(Word16 sval2shft, Word16 s_shftamount)
```

## Inline Assembly Language and Intrinsics

### *Intrinsic Functions*

---

#### Example

```
short result;  
short s1 = 0x1234;  
short s2 = 1;  
  
result = shlftNs(s1,s2);  
// Expected value of result: 0x2468
```

---

#### shlfts

Arithmetic left shift of 16-bit value by a specified shift amount. Saturation does occur during a left shift if required. When an accumulator is the destination, zeroes out the LSP portion.

---

<b>NOTE</b>	This is not a bidirectional shift.
-------------	------------------------------------

---

#### Assumptions

Assumed s\_shftamount is positive.

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

#### Prototype

```
Word16 shlfts(Word16 sval2shft, Word16 s_shftamount)
```

---

## Example

```
short result;  
short s1 = 0x1234;  
short s2 = 3;  
  
result = shlfts(s1,s2);  
// Expected value of result: 0x91a0
```

---

## shr

Arithmetic shift of 16-bit value by a specified shift amount. If the shift count is positive, a right shift is performed. Otherwise, a left shift is performed. Saturation may occur during a left shift. When an accumulator is the destination, zeroes out the LSP portion.

---

<b>NOTE</b>	This operation is not optimal on the DSP56800E because of the saturation requirements and the bidirectional capability. See the intrinsic shrtNs which is more optimal.
-------------	---

---

## Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

## Prototype

```
Word16 shr(Word16 sval2shft, Word16 s_shftamount)
```

## Example

```
short result;  
short s1 = 0x2468;  
short s2= 1;  
  
result = shr(s1,s2);  
// Expected value of result: 0x1234
```

## shr\_r

Arithmetic shift of 16-bit value by a specified shift amount. If the shift count is positive, a right shift is performed. Otherwise, a left shift is performed. If a right shift is performed, then rounding performed on result. Saturation may occur during a left shift. When an accumulator is the destination, zeroes out the LSP portion.

---

<b>NOTE</b>	This operation is not optimal on the DSP56800E because of the saturation requirements and the bidirectional capability. See the intrinsic shrtNs which is more optimal.
-------------	---

---

### Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

### Prototype

```
Word16 shr_r(Word16 s_val2shft, Word16 s_shftamount)
```

### Example

```
short result;  
short s1 = 0x2468;  
short s2= 1;  
  
result = shr(s1,s2);  
// Expected value of result: 0x1234
```

---

## shrtNs

Arithmetic shift of 16-bit value by a specified shift amount. If the shift count is positive, a right shift is performed. Otherwise, a left shift is performed. Saturation does not occur during a left shift. When an accumulator is the destination, zeroes out the LSP portion.

---

**NOTE** Ignores upper N-5 bits of s\_shftamount except the sign bit (MSB).  
If s\_shftamount is positive and the value in the lower 5 bits of s\_shftamount is greater than 15, the result is 0 if sval2shft is positive, and 0xFFFF if sval2shft is negative.  
If s\_shftamount is negative and the absolute value in the lower 5 bits of s\_shftamount is greater than 15, the result is 0.

---

## Prototype

Word16 shrtns(Word16 sval2shft, Word16 s\_shftamount)

## Example

```
short result;
short s1 = 0x2468;
short s2 = 1;

result = shrtns(s1,s2);
// Expected value of result: 0x1234
```

## L\_shl

Arithmetic shift of 32-bit value by a specified shift amount. If the shift count is positive, a left shift is performed. Otherwise, a right shift is performed. Saturation may occur during a left shift. When an accumulator is the destination, zeroes out the LSP portion.

---

**NOTE** This operation is not optimal on the DSP56800E because of the saturation requirements and the bidirectional capability. See the intrinsic L\_shlftNs or L\_shlfts which are more optimal.

---

## Assumptions

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

## Inline Assembly Language and Intrinsic Functions

---

### Prototype

```
Word32 L_shl(Word32 lval2shft, Word16 s_shftamount)
```

### Example

```
long result, l = 0x12345678;
short s2 = 1;

result = L_shl(l,s2);
// Expected value of result: 0x2468ACF0
```

---

## L\_shlftNs

Arithmetic shift of 32-bit value by a specified shift amount. If the shift count is positive, a left shift is performed. Otherwise, a right shift is performed. Saturation does not occur during a left shift.

---

<b>NOTE</b>	Ignores upper N-5 bits of s_shftamount except the sign bit (MSB).
-------------	---

---

### Prototype

```
Word32 L_shlftNs(Word32 lval2shft, Word16 s_shftamount)
```

### Example

```
long result, l = 0x12345678;
short s2= 1;

result = L_shlftNs(l,s2);
// Expected value of result: 0x2468ACF0
```

---

## L\_shlfts

Arithmetic left shift of 32-bit value by a specified shift amount. Saturation does occur during a left shift if required.

---

**NOTE** This is not a bidirectional shift.

---

### Assumptions

Assumed s\_shftamount is positive.

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

### Prototype

```
Word32 L_shlfts(Word32 lval2shft, Word16 s_shftamount)
```

### Example

```
long result, l = 0x12345678;
short s1 = 3;

result = shlfts(l, s1);
// Expected value of result: 0x91A259E0
```

---

## L\_shr

Arithmetic shift of 32-bit value by a specified shift amount. If the shift count is positive, a right shift is performed. Otherwise, a left shift is performed. Saturation may occur during a left shift. When an accumulator is the destination, zeroes out the LSP portion.

---

**NOTE** This operation is not optimal on the DSP56800E because of the saturation requirements and the bidirectional capability. See the intrinsic L\_shrtNs which is more optimal.

---

## Inline Assembly Language and Intrinsics

### *Intrinsic Functions*

---

#### **Assumptions**

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

#### **Prototype**

```
Word32 L_shr(Word32 lval2shft, Word16 s_shftamount)
```

#### **Example**

```
long result, l = 0x24680000;
short s2= 1;

result = L_shrtNs(l,s2);
// Expected value of result: 0x12340000
```

---

## **L\_shr\_r**

Arithmetic shift of 32-bit value by a specified shift amount. If the shift count is positive, a right shift is performed. Otherwise, a left shift is performed. If a right shift is performed, then rounding performed on result. Saturation may occur during a left shift.

#### **Assumptions**

OMR's SA bit was set to 1 at least 3 cycles before this code, that is, saturation on data ALU results enabled.

#### **Prototype**

```
Word32 L_shr_r(Word32 lval2shft, Word16 s_shftamount)
```



---

## Example

```
long l1 = 0x41111111;  
short s2 = 1;  
long result;  
  
result = L_shr_r(l1,s2);  
// Expected value of result: 0x20888889
```

---

## L\_shrtNs

Arithmetic shift of 32-bit value by a specified shift amount. If the shift count is positive, a right shift is performed. Otherwise, a left shift is performed. Saturation does not occur during a left shift.

---

<b>NOTE</b>	Ignores upper N-5 bits of s_shftamount except the sign bit (MSB).
-------------	---

---

## Prototype

```
Word32 L_shrtNs(Word32 lval2shft, Word16 s_shftamount)
```

## Example

```
long result, l = 0x24680000;  
short s2= 1;  
  
result = L_shrtNs(l,s2);  
// Expected value of result: 0x12340000
```


## Modulo Addressing Intrinsic Functions

A modulo buffer is a buffer in which the data pointer loops back to the beginning of the buffer once the pointer address value exceeds a specified limit.

Figure 11.1 depicts a modulo buffer with the limit six. Increasing the pointer address value to 0x106 makes it point to the same data it would point to if its address value were 0x100.

**Figure 11.1 Example of a Modulo Buffer**

Address	Data
0x100	0.68
0x101	0.73
0x102	0.81
0x103	0.86
0x104	0.90
0x105	0.95



The CodeWarrior C compiler for DSP56800E uses intrinsic functions to create and manipulate modulo buffers. Normally, a modulo operation, such as the % operator, requires a runtime function call to the arithmetic library. For normally timed critical DSP loops, this binary operation imposes a large execution-time overhead.

The CodeWarrior implementation, however, replaces the runtime call with an efficient implementation of circular-address modification, either by using hardware resources or by manipulating the address mathematically.

Processors such as the DSP56800E have on-chip hardware support for modulo buffers. Modulo control registers work with the DSP pointer update addressing modes to access a range of addresses instead of a continuous, linear address space. But hardware support imposes strict requirements on buffer address alignment, pointer register resources, and limited modulo addressing instructions. For example, R0 and R1 are the only registers available for modulo buffers.

Accordingly, the CodeWarrior C compiler uses a well-defined set of intrinsic APIs to implement modulo buffers.

## Modulo Addressing Intrinsic Functions

The intrinsic functions for modulo addressing are:

- `__mod_init`
- `__mod_initint16`
- `__mod_start`
- `__mod_access`
- `__mod_update`
- `__mod_stop`
- `__mod_getint16`

- `__mod_setint16`
- `__mod_error`

---

## `__mod_init`

Initialize a modulo buffer pointer with arbitrary data using the address specified by the `<addr_expr>`. This function expects a byte address. `<addr_expr>` is an arbitrary C expression which normally evaluates the address at the beginning of the modulo buffer, although it may be any legal buffer address. The `<mod_desc>` evaluates to a compile time constant of either 0 or 1, represented by the modulo pointers R0 or R1, respectively. The `<mod_sz>` is a compile time integer constant representing the size of the modulo buffer in bytes. The `<data_sz>` is a compile time integer constant representing the size of data being stored in the buffer in bytes. `<data_sz>` is usually derived from the `sizeof()` operator.

The `__mod_init` function may be called independently for each modulo pointer register.

If `__mod_error` has not been previously called, no record of `__mod_init` errors are saved.

If `__mod_error` has been previously called, `__mod_init` may set one of the error condition in the static memory location defined by `__mod_error`. (See `__mod_error` description for a complete list of error conditions).

### Prototype

```
void __mod_init (  
    int <mod_desc>,  
    void * <addr_expr>,  
    int <mod_sz>,  
    int <data_sz> );
```

### Example

Initialize a modulo buffer pointer with a buffer size of 3 and where each element is a structure:

## Inline Assembly Language and Intrinsics *Intrinsic Functions*

---

```
__mod_init(0, (void *)&struct_buf[0], 3, sizeof(struct  
mystruct) );
```

---

### **\_\_mod\_initint16**

Initialize modulo buffer pointer with integer data. The \_\_mod\_initint16 function behaves similarly to the \_\_mod\_init function, except that word addresses are used to initialize the modulo pointer register.

#### **Prototype**

```
void __mod_initint16(  
int <mod_desc>,  
int * <addr_expr>,  
int <mod_sz> );
```

#### **Example**

Initialize an integer modulo buffer pointer with a buffer size of 10.

```
__mod_initint16(0, &int_buf[9], 10);
```

---

### **\_\_mod\_start**

Write the modulo control register. The \_\_mod\_start function simply writes the modulo control register (M01) for each modulo pointer register which has been previously initialized. The values written to M01 depends on the size of the modulo buffer and which pointers have been initialized.

#### **Prototype**

```
void __mod_start( void );
```

---

---

## **\_\_mod\_access**

Retrieve the modulo pointer. The `__mod_access` function returns the modulo pointer value specified by `<mod_desc>` in the R2 register, as per calling conventions. The value returned is a byte address. The data in the modulo buffer may be read or written by a cast and dereference of the resulting pointer.

### **Prototype**

```
void *__mod_access( int <mod_desc> );
```

### **Example**

Assign a value to the modulo buffer at the current pointer.

```
((char *)__mod_access(0)) = (char)i;
```

---

## **\_\_mod\_update**

Update the modulo pointer. The `__mod_update` function updates the modulo pointer by the number of data type units specified in `<amount>`. `<amount>` may be negative. Of course, the pointer will wrap to the beginning of the modulo buffer if the pointer is advanced beyond the modulo boundaries. `<amount>` must be a compile time constant.

### **Prototype**

```
void __mod_update( int <mod_desc>, int <amount> );
```

### **Example**

Advance the modulo pointer by 2 units.

```
__mod_update(0, 2);
```

---

## **\_\_mod\_stop**

Reset modulo addressing to linear addressing. This function writes the modulo control register with a value which restore linear addressing to the R0 and R1 pointer registers.

## Prototype

```
void __mod_stop( int <mod_desc> );
```

---

## **\_\_mod\_getint16**

Retrieve a 16-bit signed value from the modulo buffer and update the modulo pointer. This function returns an integer value from the location pointed to by the modulo pointer. The function then updates the modulo pointer by <amount> integer units (<amount>\*2 bytes). <amount> must be a compile time constant.

## Prototype

```
int __mod_getint16( int <mod_desc>, int <amount> );
```

## Example

Retrieve an integer value from a modulo buffer and update the modulo buffer pointer by one word.

```
int y;  
y = __mod_getint16(0, 1);
```

---

## **\_\_mod\_setint16**

Write a 16-bit signed integer to the modulo buffer and update the pointer. This function evaluates <int\_expr> and copies the value to the location pointed to by the modulo pointer. The modulo pointer is then updated by <amount>. <amount> must be a compile time constant.

## Prototype

```
int __mod_setint16( int <mod_desc>, int <int_expr>, int <amount>  
);
```

## Example

Write the modulo buffer with a value derived from an expression, do not update modulo pointer.

---

---

```
__mod_setint16( 0, getrandomint(), 0 );
```

---

### **\_\_mod\_error**

Set up a modulo error variable. This function registers a static integer address to hold the error results from any of the modulo buffer API calls. The function returns 0 if it is successful, 1 otherwise. The argument must be the address of a static, global integer variable. This variable holds the result of calling each of the previously defined API functions. This allows the user to monitor the status of the error variable and take action if the error variable is non-zero. Typically, the user would use \_\_mod\_error during development and remove it once debugging is complete. \_\_mod\_error generates no code, although the error variable may occupy a word of memory. A non-zero value in the error variable indicates a misuse of the one of the API functions. Once the error variable is set it is reset when \_\_mod\_stop is called. The error variable contains the error number of the last error. A successful call to an API function will not reset the error variable; only \_\_mod\_stop will reset the error variable.

#### **Prototype**

```
int __mod_error( int * <static_object_addr> );
```

#### **Example**

Register the error number variable

```
static int myerrno;

assert( __mod_error(&myerrno) == 0 ) ;
```

### **Modulo Buffer Examples**

Listing 11.11 and Listing 11.12 are two modulo buffer examples.

#### **Listing 11.11 Modulo Buffer Example 1**

---

```
#pragma define_section DATA_INT_MODULO ".data_int_modulo"

/* Place the buffer object in a unique section so the it can be aligned
properly in the linker control file. */
```

---

# Freescale Semiconductor, Inc.

## Inline Assembly Language and Intrinsics

### *Intrinsic Functions*

---

```
#pragma section DATA_INT_MODULO begin
int int_buf[10];
#pragma section DATA_INT_MODULO end

/* Convenient defines for modulo descriptors */

#define M0 0
#define M1 1

int main ( void )
{
    int i;

    /* Modulo buffer will be initialized. R0 will be the modulo pointer
    register. The buffer size is 10 units. The unit size is 'sizeof(int)'.
    */

    __mod_init(M0, (void *)&int_buf[0], 10, sizeof(int));

    /* Write the modulo control register */

    __mod_start();

    /* Write int_buf[0] through int_buf[9]. R0 initially points at
    int_buf[0] and wraps when the pointer value exceeds int_buf[9]. The
    pointer is updated by 1 unit each time through the loop */

    for ( i=0; i<100; i++ )
    {

                                                                    *((int *)__mod_access(M0))
= i;                                                                    __mod_update(M0, 1);

    }

    /* Reset modulo control register to linear addressing mode */
    __mod_stop();
}
```

---



---

## Listing 11.12 Modulo Buffer Example 2

---

```
/* Set up a static location to save error codes */
if ( ! __mod_error(&err_codes)) {
    printf ("__mod_error set up failed\n");
}

/* Initialize a modulo buffer pointer, pointing to an array of 10 ints.
*/

__mod_initint16(M0, &int_buf[9], 10);

/* Check for success of previous call */

if ( err_code ) { printf (
    "__mod_initint16 failed\n" ) };

__mod_start();

/* Write modulo buffer with the result of the expression "i".
Decrement the buffer pointer for each execution of the loop.
The modulo buffer wraps from index 0 to 9 through the entire execution
of the loop. */

for ( i=100; i>0; i-- ) {
    __mod_setint16(M0, i, -1);
}
__mod_stop();
```

---

## Points to Remember

As you use modulo buffer intrinsic functions, keep these points in mind:

1. You must align modulo buffers properly, per the constraints that the *M56800E User's Manual* explains. There is no run-time validation of alignment. Using the modulo buffer API on unaligned buffers will cause erratic, unpredictable behavior during data accesses.
2. Calling `__mod_start()` to write to the modulo control register effectively changes the hardware's global-address-generation state. This change of state

affects all user function calls, run-time supporting function calls, standard library calls, and interrupts.

3. You must account for any side-effects of enabling modulo addressing. Such a side-effect is that R0 and R1 update in a modulo way.
4. If you need just one modulo pointer is required, use the R0 address register. Enabling the R1 address register for modulo use also enables the R0 address register for modulo use. This is true even if `__mod_init()` or `__mod_initint16()` have not explicitly initialized R0.
5. A successful API call does not clear the error code from the error variable. Only function `__mod_stop` clears the error code.

## Modulo Addressing Error Codes

If you want to register a static variable for error-code storage, use `__mod_error()`. In case of an error occur, this static variable will contain one of the values Table 11.3 explains. Table 11.4. lists the error codes possible for each modulo addressing intrinsic function.

**Table 11.3 Modulo Addressing Error Codes**

Code	Meaning
11	<mod_desc> parameter must be zero or one.
12	R0 modulo pointer is already initialized. An extraneous call to <code>__mod_init</code> or <code>__mod_initint16</code> to initialize R0 has been made.
13	R1 modulo pointer is already initialized. An extraneous call to <code>__mod_init</code> or <code>__mod_initint16</code> to initialize R1 has been made.
14	Modulo buffer size must be a compile time constant.
15	Modulo buffer size must be greater than one.
16	Modulo buffer size is too big.
17	Modulo buffer size for R0 and R1 must be the same.
18	Modulo buffer data types for R0 and R1 must be the same.
19	Modulo buffer has not been initialized.
20	Modulo buffer has not been started.
21	Parameter is not a compile time constant.

**Table 11.3 Modulo Addressing Error Codes (*continued*)**

Code	Meaning
22	Attempt to use word pointer functions with byte pointer initialization. <code>__mod_getint16</code> and <code>__mod_setint16</code> were called but <code>__mod_init</code> was used for initialization. <code>__mod_initint16</code> is required for pointer initialization.
23	Modulo increment value exceeds modulo buffer size.
24	Attempted use of R1 as a modulo pointer without initializing R0 for modulo use.

**Table 11.4 Possible Error Codes**

Function	Possible Error Code
<code>__mod_init</code>	11, 12, 13, 14, 15, 16, 17, 18, 21
<code>__mod_stop</code>	none
<code>__mod_getint16</code>	11, 14, 20, 22, 24
<code>__mod_setint16</code>	11, 14, 20, 22, 24
<code>__mod_start</code>	none
<code>__mod_access</code>	11, 19, 20, 24
<code>__mod_update</code>	11, 14, 20, 23, 24
<code>__mod_initint16</code>	11, 12, 13, 14, 15, 16, 17



## ELF Linker

---

The CodeWarrior™ Executable and Linking Format (ELF) Linker makes a program file out of the object files of your project. The linker also allows you to manipulate code in different ways. You can define variables during linking, control the link order to the granularity of a single function, change the alignment, and even compress code and data segments so that they occupy less space in the output file.

All of these functions are accessed through commands in the linker command file (LCF). The linker command file has its own language complete with keywords, directives, and expressions, that are used to create the specifications for your output code. The syntax and structure of the linker command file is similar to that of a programming language.

This chapter contains the following sections:

- Structure of Linker Command Files
- Linker Command File Syntax
- Linker Command File Keyword Listing

## Structure of Linker Command Files

Linker command files contain three main segments:

- Memory Segment
- Closure Blocks
- Sections Segment

A command file must contain a memory segment and a sections segment. Closure segments are optional.

## Memory Segment

In the memory segment, available memory is divided into segments. The memory segment format looks like Listing 12.1.

---

## Listing 12.1 Sample MEMORY Segment

---

```
MEMORY {  
    segment_1 (RWX): ORIGIN = 0x8000, LENGTH = 0x1000  
    segment_2 (RWX): ORIGIN = AFTER(segment_1), LENGTH = 0  
    data      (RW) : ORIGIN = 0x2000, LENGTH = 0x0000  
    #segment_name (RW) : ORIGIN = memory address, LENGTH = segment  
length  
    #and so on...  
}
```

---

The first memory segment definition (`segment_1`) can be broken down as follows:

- the (RWX) portion of the segment definition pertains to the ELF access permission of the segment. The (RWX) flags imply **read**, **write**, and **execute** access.
- ORIGIN represents the start address of the memory segment (in this case 0x8000).
- LENGTH represents the size of the memory segment (in this case 0x1000).

Memory segments with RWX attributes are placed in to P: memory while RW attributes are placed into X: memory.

If you cannot predict how much space a segment will occupy, you can use the function AFTER and LENGTH = 0 (unlimited length) to fill in the unknown values.

## Closure Blocks

The linker is very good at deadstripping unused code and data. Sometimes, however, symbols need to be kept in the output file even if they are never directly referenced. Interrupt handlers, for example, are usually linked at special addresses, without any explicit jumps to transfer control to these places.

Closure blocks provide a way to make symbols immune from deadstripping. The closure is transitive, meaning that symbols referenced by the symbol being closed are also forced into closure, as are any symbols referenced by those symbols, and so on.

---

<b>NOTE</b>	The closure blocks need to be in place before the SECTIONS definition in the linker command file.
-------------	---

---

The two types of closure blocks available are:

- Symbol-level

---

Use `FORCE_ACTIVE` to include a symbol into the link that would not be otherwise included. An example is shown in Listing 12.2.

## Listing 12.2 Sample Symbol-level Closure Block

---

```
FORCE_ACTIVE {break_handler, interrupt_handler, my_function}
```

---

- Section-level

Use `KEEP_SECTION` when you want to keep a section (usually a user-defined section) in the link. Listing 12.3 shows an example.

## Listing 12.3 Sample Section-level Closure Block

---

```
KEEP_SECTION { .interrupt1, .interrupt2 }
```

---

A variant is `REF_INCLUDE`. It keeps a section in the link, but only if the file where it is coming from is referenced. This is very useful to include version numbers. Listing 12.4 shows an example of this.

## Listing 12.4 Sample Section-level Closure Block With File Dependency

---

```
REF_INCLUDE { .version }
```

---

## Sections Segment

Inside the sections segment, you define the contents of your memory segments, and define any global symbols to be used in the output file.

The format of a typical sections block looks like Listing 12.5.

---

<b>NOTE</b>	As shown in Listing 12.5, the <code>.bss</code> section always needs to be put at the end of a segment or in a standalone segment, because it is not a loadable section.
-------------	--

---

## Listing 12.5 Sample SECTIONS Segment

---

```
SECTIONS {  
    .section_name : #the section name is for your reference  
    {  
        #the section name must begin with a '.'  
        filename.c (.text) #put the .text section from filename.c  
    }
```

---

## ELF Linker

### Linker Command File Syntax

---

```
filename2.c (.text) #then the .text section from filename2.c
filename.c (.data)
filename2.c (.data)
filename.c (.bss)
filename2.c (.bss)
. = ALIGN (0x10); #align next section on 16-byte boundary.
} > segment_1 #this means "map these contents to segment_1"

.next_section_name:
{
    more content descriptions
} > segment_x # end of .next_section_name definition
} # end of the sections block
```

---

## Linker Command File Syntax

This section explains some practical ways in which to use the commands of the linker command file to perform common tasks.

### Alignment

To align data on a specific byte-boundary, use the ALIGN and ALIGNALL commands to bump the location counter to the preferred boundary. For example, the following fragment uses ALIGN to bump the location counter to the next 16-byte boundary. An example is given in Listing 12.6.

#### Listing 12.6 Sample ALIGN Command Usage

---

```
file.c (.text)
. = ALIGN (0x10);
file.c (.data) # aligned on a 16-byte boundary.
```

---

You can also align data on a specific byte-boundary with ALIGNALL, as shown in Listing 12.7.

#### Listing 12.7 Sample ALIGNALL Command Usage

---

```
file.c (.text)
ALIGNALL (0x10); #everything past this point aligned on 16 bytes
file.c (.data)
```

---



## Arithmetic Operations

Standard C arithmetic and logical operations may be used to define and use symbols in the linker command file. Table 12.1 shows the order of precedence for each operator. All operators are left-associative.

**Table 12.1 Arithmetic Operators**

Precedence	Operators
highest (1)	- ~ !
2	* / %
3	+ -
4	>> <<
5	== != > < <= >=
6	&
7	
8	&&
9	

**NOTE** The shift operator shifts two-bits for each shift operation. The divide operator performs division and rounding.

## Comments

Comments may be added by using the pound character (#) or C++ style double-slashes (/ /). C-style comments are not accepted by the LCF parser. Listing 12.8 shows examples of valid comments.

**Listing 12.8 Sample Comments**

```
# This is a one-line comment
* (.text) // This is a partial-line comment
```

## Deadstrip Prevention

The M56800E linker removes unused code and data from the output file. This process is called deadstripping. To prevent the linker from deadstripping unreferenced code and data, use the `FORCE_ACTIVE`, `KEEP_SECTION`, and `REF_INCLUDE` directives to preserve them in the output file.

## Variables, Expressions, and Integral Types

This section explains variables, expressions, and integral types.

### Variables and Symbols

All symbol names within a Linker Command File (LCF) start with the underscore character (`_`), followed by letters, digits, or underscore characters. Listing 12.9 shows examples of valid lines for a command file:

**Listing 12.9 Valid Command File Lines**

---

```
_dec_num = 99999999;  
_hex_num = 0x9011276;
```

---

Variables that are defined within a `SECTIONS` section can only be used within a `SECTIONS` section in a linker command file.

### Global Variables

Global variables are accessed in a linker command file with an `'F'` prepended to the symbol name. This is because the compiler adds an `'F'` prefix to externally defined symbols.

Listing 12.10 shows an example of using a global variable in a linker command file. This example sets the global variable `_foot`, declared in C with the `extern` keyword, to the location of the address location current counter.

**Listing 12.10 Using a Global Variable in the LCF**

---

```
F_foot = .;
```

---

If you use a global symbol in an LCF, as in Listing 12.10, you can access it from C program sources as shown in Listing 12.11.

---

## Listing 12.11 Accessing a Global Symbol From C Program Sources

---

```
extern unsigned long _foot;
void main( void ) {
    unsigned long i;
    // ...
    i = _foot;    // _foot value determined in LCF
    // ...
}
```

---

## Expressions and Assignments

You can create symbols and assign addresses to those symbols by using the standard assignment operator. An assignment may only be used at the start of an expression, and a semicolon is required at the end of an assignment statement. An example of standard assignment operator usage is shown in Listing 12.12.

---

## Listing 12.12 Standard Assignment Operator Usage

---

```
_symbolicname = some_expression;           # Legal
_sym1 + _sym2 = _sym3;                        # ILLEGAL!
```

---

When an expression is evaluated and assigned to a variable, it is given either an absolute or a relocatable type. An absolute expression type is one in which the symbol contains the value that it will have in the output file. A relocatable expression is one in which the value is expressed as a fixed offset from the base of a section.

## Integral Types

The syntax for linker command file expressions is very similar to the syntax of the C programming language. All integer types are long or unsigned long.

Octal integers (commonly known as base eight integers) are specified with a leading zero, followed by numeral in the range of zero through seven. Listing 12.13 shows valid octal patterns that you can put into your linker command file.

---

## Listing 12.13 Sample Octal Patterns

---

```
_octal_number  = 012;
_octal_number2 = 03245;
```

---

## ELF Linker

### Linker Command File Syntax

---

Decimal integers are specified as a non-zero numeral, followed by numerals in the range of zero through nine. To create a negative integer, use the minus sign (-) in front of the number. Listing 12.14 shows examples of valid decimal integers that you can write into your linker command file.

#### Listing 12.14 Sample Decimal Integers

---

```
_dec_num      = 9999;  
_decimalNumber = -1234;
```

---

Hexadecimal (base sixteen) integers are specified as 0x or 0X (a zero with an X), followed by numerals in the range of zero through nine, and/or characters A through F. Examples of valid hexadecimal integers that you can put in your linker command file appear in Listing 12.15.

#### Listing 12.15 Sample Hex Integers

---

```
_somenumber    = 0x0F21;  
_fudgefactorSpace = 0XF00D;  
_hexonyou      = 0xcafe;
```

---

---

<b>NOTE</b>	When assigning a value to a pointer variable, the value is in byte units despite that in the linked map (.xMAP file), the variable value appears in word units.
-------------	---

---

## File Selection

When defining the contents of a SECTION block, specify the source files that are contributing to their sections.

In a large project, the list can become very long. For this reason, you have to use the asterisk (\*) keyword. The \* keyword represents the filenames of every file in your project. Note that since you have already added the .text sections from the main.c, file2.c, and file3.c files, the \* keyword does not include the .text sections from those files again.

---

## Function Selection

The OBJECT keyword allows precise control over how functions are placed within a section. For example, if the functions `pad` and `foot` are to be placed before anything else in a section, use the code as shown in the example in Listing 12.16.

### Listing 12.16 Sample Function Selection Using OBJECT Keyword

```
SECTIONS {  
  .program_section :  
  {  
    OBJECT (Fpad, main.c)  
    OBJECT (Ffoot, main.c)  
    * (.text)  
  } > ROOT  
}
```

---

<b>NOTE</b>	If an object is written once using the OBJECT function selection keyword, the same object will not be written again if you use the '*' file selection keyword.
-------------	--

---

## ROM to RAM Copying

In embedded programming, it is common to copy a portion of a program resident in ROM into RAM at runtime. For example, program variables cannot be accessed until they are copied to RAM.

To indicate data or code that is meant to be copied from ROM to RAM, the data or code is assigned two addresses. One address is its resident location in ROM (where it is downloaded). The other is its intended location in RAM (where it is later copied in C code).

Use the MEMORY segment to specify the intended RAM location, and the AT (address) parameter to specify the resident ROM address.

For example, you have a program and you want to copy all your initialized data into RAM at runtime. Listing 12.17 shows the LCF you use to set up for writing data to ROM.

## ELF Linker

### Linker Command File Syntax

---

#### Listing 12.17 LCF to Setup for ROM to RAM Copy

---

```
MEMORY {
    .text (RWX) : ORIGIN = 0x8000, LENGTH = 0x0    # code (p:)
    .data (RW)   : ORIGIN = 0x3000, LENGTH = 0x0    # data (x:)-> RAM
}

SECTIONS{

    .main_application :
    {
        # .text sections

        *(.text)
        *(.rtlib.text)
        *(.fp_engine.txt)
        *(user.text)
    } > .text

    __ROM_Address = 0x2000
    .data : AT(__ROM_Address) # ROM Address definition
    {
        # .data sections
        F_Begin_Data = .;      # Start location for RAM (0x3000)
        *(.data)               # Write data to the section (ROM)
        *(fp_state.data);
        *(rtlib.data);
        F_End_Data = .;        # Get end location for RAM

        # .bss sections
        * (rtlib.bss.lo)
        * (.bss)
        F_ROM_Address = __ROM_Address
    } > .data
}
```

---

To make the runtime copy from ROM to RAM, you need to know where the data starts in ROM (`__ROM_Address`) and the size of the block in ROM you want to copy to RAM. In the following example (Listing 12.18), copy all variables in the data section from ROM to RAM in C code.

---

**Listing 12.18 ROM to RAM Copy From C After Writing Data Flash**

---

```
#include <stdio.h>
#include <string.h>

int GlobalFlash = 6;

// From linker command file
extern __Begin_Data, __ROMAddress, __End_Data;

void main( void )
{
    unsigned short a = 0, b = 0, c = 0;
    unsigned long dataLen = 0x0;
    unsigned short __myArray[] = { 0xdead, 0xbeef, 0xcafe };

    // Calculate the data length of the X: memory written to Flash
    dataLen = (unsigned long)&__End_Data -
        (unsigned long)&__Begin_Data;

    // Block move from ROM to RAM
    memcpy( (unsigned long *)&__Begin_Data,
        (const unsigned long *)&__ROMAddress, dataLen );

    a = GlobalFlash;

    return;
}
```

---

## Utilizing Program Flash and Data RAM for Constant Data in C

There are many advantages and one disadvantage if constant data in C is flashed to program flash memory (pROM) and copied to data flash memory (xRAM) at startup, with the usual pROM-to-xRAM initialization.

The advantages are:

- constant data is defined and addressed conventionally via C language
- pROM flash space is used for constant data (pROM is usually larger than xROM)
- the pROM flash is now freed up or available

## ELF Linker

### Linker Command File Syntax

---

The disadvantage is that the xRAM is consumed for constant data at run-time.

If you wish to store constant data in program flash memory and have it handled by the pROM-to-xRAM startup process, a simple change is necessary to the pROM-to-xRAM LCF. Simply, place the constant data references into the `data_in_p_flash_ROM` section after the `__xRAM_data_start` variable like the other data references and remove the "data in xROM" section. Please see Listing 12.19.

#### Listing 12.19 Using the Typical pROM-to-xRAM LCF

---

```
.data_in_p_flash_ROM : AT(__pROM_data_start)
{
    __xRAM_data_start = .;

    * (.const.data.char) # move constant data references here
    * (.const.char)

    * (.data.char)
    * (.data)

    etc.
}
```

---

## Utilizing Program Flash for User-Defined Constant Section in Assembler

There are many advantages and one disadvantage in writing specific data to pROM with linker commands and accessing this data in assembly,

The advantages are:

- pROM flash space is used for user-specified constant data (pROM is usually larger than xROM), where the constant data is defined and addressed by assembly language
- part of the pROM flash is now freed up or available

The disadvantage is that data is not defined or accessed conventionally via C language; data is specifically flashed to pROM via the linker command file and fetched from pROM with assembly.

If you want to keep specific constant data in pROM and access it from there, you can use the linker commands to explicitly store the data in pROM and then later access the data in pROM with assembly.

The next two sections describe putting data in the pROM flash at build and run-time.



---

## Putting Data in pROM Flash at Build-time

The linker commands have specific instructions which set values in the binary image at the build time (Listing 12.20). For example, WRITEH inserts two bytes of data at the current address of a section. These commands are placed in the LCF, which tells the linker at build time to place data in P or X memory. Optionally, you can also set the current location prior to the write command to ensure a specific location address for easier reference later. The location within the section is not important.

For more information, please see the LCF section in this document.

### Listing 12.20 LCF write example using MC56F832x for build-time

---

```
.executing_code :
{
    # .text sections

    . = 0x00A4; # optionally set the location -- we use 0x00A4 in this
case
WRITEH(0xABCD); # now set some value here; location within the
section is not important
* (.text)
* (interrupt_routines.text)
* (rtlib.text)
* (fp_engine.text)
* (user.text)

etc

} > .p_flash_ROM
```

---

## Putting Data in pROM Flash at Run-time

The assembly example in Listing 12.21 fetches the pROM-flashed value at run-time in Listing 12.20.

### Listing 12.21 LCF write example using MC56F832x for run-time

---

```
move.l #$00A4, r1 ; move the pROM address into r3
move.w p:(r3)+, x0 ; fetch data from pROM at address r1 into x0
```

---

## ELF Linker

### Linker Command File Keyword Listing

---

## Stack and Heap

To reserve space for the stack and heap, arithmetic operations are performed to set the values of the symbols used by the runtime.

The Linker Command File (LCF) performs all the necessary stack and heap initialization. When Stationery is used to create a new project, the appropriate LCFs are added to the new project.

See any Stationery-generated LCFs for examples of how stack and heap are initialized.

## Writing Data Directly to Memory

You can write data directly to memory using the `WRITEx` command in the linker command file. The `WRITEB` command writes a byte, the `WRITEH` command writes two bytes, and the `WRITEW` command writes four bytes. You insert the data at the section's current address.

### Listing 12.22 Embedding Data Directly Into Output

---

```
.example_data_section :  
{  
    WRITEB 0x48;    // 'H'  
    WRITEB 0x69;    // 'i'  
    WRITEB 0x21;    // '!'  
}
```

---

## Linker Command File Keyword Listing

This section explains the keywords available for use when creating CodeWarrior Development Studio for Freescale 56800/E Hybrid Controllers application objects with the linker command file. Valid linker command file functions, keywords, directives, and commands are:

---

### . (location counter)

The period character ( `.` ) always maintains the current position of the output location. Since the period always refers to a location in a `SECTIONS` block, it can not be used outside a section definition.

A period may appear anywhere a symbol is allowed. Assigning a value to period that is greater than its current value causes the location counter to move, but the location counter can never be decremented.

This effect can be used to create empty space in an output section. In the example below, the location counter is moved to a position that is 0x1000 words past the symbol FSTART\_.

### Example

```
.data :
{
    *(.data)
    *(.bss)
    FSTART_ = .;
    . = FSTART_ + 0x1000;
    __end = .;
} > DATA
```

---

## ADDR

The ADDR function returns the address of the named section or memory segment.

### Prototype

ADDR (*sectionName* | *segmentName* | *symbol*)

In the example below, ADDR is used to assign the address of ROOT to the symbol \_\_rootbasecode.

### Example

```
MEMORY{
    ROOT (RWX) : ORIGIN = 0x8000, LENGTH = 0
}

SECTIONS{
    .code :
    {
```

## ELF Linker

### Linker Command File Keyword Listing

---

```
__rootbasecode = ADDR(ROOT);  
    *(.text);  
} > ROOT  
}
```

---

<b>NOTE</b>	In order to use segmentName with this command, the segmentName must start with the period character even though segmentNames are not required to start with the period character by the linker, as is the case with sectionName.
-------------	--

---

---

## ALIGN

The **ALIGN** function returns the value of the location counter aligned on a boundary specified by the value of `alignValue`. The `alignValue` must be a power of two.

### Prototype

`ALIGN(alignValue)`

Note that **ALIGN** does not update the location counter; it only performs arithmetic. To update the location counter, use an assignment such as:

### Example

```
. = ALIGN(0x10);    #update location counter to 16  
                   #byte alignment
```

---

## ALIGNALL

**ALIGNALL** is the command version of the **ALIGN** function. It forces the minimum alignment for all the objects in the current segment to the value of `alignValue`. The `alignValue` must be a power of two.

### Prototype

`ALIGNALL(alignValue);`

Unlike its counterpart `ALIGN`, `ALIGNALL` is an actual command. It updates the location counter as each object is written to the output.

## Example

```
.code :
{
    ALIGNALL(16); // Align code on 16 byte boundary
    *      (.init)
    *      (.text)

    ALIGNALL(16); //align data on 16 byte boundary
    *      (.rodata)
} > .text
```

---

## FORCE\_ACTIVE

The `FORCE_ACTIVE` directive allows you to specify symbols that you do not want the linker to deadstrip. You must specify the symbol(s) you want to keep before you use the `SECTIONS` keyword.

### Prototype

```
FORCE_ACTIVE{ symbol[, symbol] }
```

---

## INCLUDE

The `INCLUDE` command let you include a binary file in the output file.

### Prototype

```
INCLUDE filename
```

## ELF Linker

### Linker Command File Keyword Listing

---

## KEEP\_SECTION

The `KEEP_SECTION` directive allows you to specify sections that you do not want the linker to deadstrip. You must specify the section(s) you want to keep before you use the `SECTIONS` keyword.

### Prototype

```
KEEP_SECTION{ sectionType[, sectionType] }
```

---

## MEMORY

The `MEMORY` directive allows you to describe the location and size of memory segment blocks in the target. This directive specifies the linker the memory areas to avoid, and the memory areas into which it links the code and data.

The linker command file may only contain one `MEMORY` directive. However, within the confines of the `MEMORY` directive, you may define as many memory segments as you wish.

### Prototype

```
MEMORY { memory_spec }
```

The *memory\_spec* is:

*segmentName* (*accessFlags*) : *ORIGIN* = *address*, *LENGTH* = *length*, [*COMPRESS*]  
[> *fileName*]

*segmentName* can include alphanumeric characters and underscore '\_' characters.

*accessFlags* are passed into the output ELF file (`Phdr.p_flags`). The *accessFlags* can be:

- R-read
- W-write
- X-executable (for P: memory placement)

*ORIGIN address* is one of the following:

a memory address	Specify a hex address, such as 0x8000.
an AFTER command	Use the AFTER(name [,name]) command to tell the linker to place the memory segment after the specified segment. In the example below, overlay1 and overlay2 are placed after the code segment. When multiple memory segments are specified as parameters for AFTER, the highest memory address is used.

## Example

```
memory{
  code      (RWX)  : ORIGIN = 0x8000,      LENGTH = 0
  overlay1  (RWX)  : ORIGIN = AFTER(code), LENGTH = 0
  overlay2  (RWX)  : ORIGIN = AFTER(code), LENGTH = 0
  data      (RW)   : ORIGIN = 0x1000,      LENGTH = 0
}
```

ORIGIN is the assigned address.

LENGTH is one of the following:

a value greater than zero	If you try to put more code and data into a memory segment than your specified length allows, the linker stops with an error.
autolength by specifying zero	When the length is 0, the linker lets you put as much code and data into a memory segment as you want.

---

**NOTE**      There is no overflow checking with autolength. The linker can produce an unexpected result if you use the autolength feature without leaving enough free memory space to contain the memory segment. For this reason, when you use autolength, use the AFTER keyword to specify origin addresses.

---

> fileName is an option to write the segment to a binary file on disk instead of an ELF program header. The binary file is put in the same folder as the ELF output file. This option has two variants:

>fileName	Writes the segment to a new file.
>>fileName	Appends the segment to an existing file.

## ELF Linker

Linker Command File Keyword Listing

---

## OBJECT

The OBJECT keyword allows control over the order in which functions are placed in the output file.

### Prototype

```
OBJECT (function, sourcefile.c)
```

It is important to note that if you write an object to the output file using the OBJECT keyword, the same object will not be written again by either the GROUP keyword or the '\*' wildcard.

---

## REF\_INCLUDE

The REF\_INCLUDE directive allows you to specify sections that you do not want the linker to deadstrip, but only if they satisfy a certain condition: the file that contains the section must be referenced. This is useful if you want to include version information from your source file components. You must specify the section(s) you want to keep before you use the SECTIONS keyword.

### Prototype

```
REF_INCLUDE{ sectionType [, sectionType] }
```

---

## SECTIONS

A basic SECTIONS directive has the following form:

### Prototype

```
SECTIONS { <section_spec> }
```

*section\_spec* is one of the following:

- *sectionName*: [AT (*loadAddress*)] { contents } > *segmentName*
- *sectionName*: [AT (*loadAddress*)] { contents } >> *segmentName*



`sectionName` is the section name for the output section. It must start with a period character. For example, `".mysection"`.

`AT (loadAddress)` is an optional parameter that specifies the address of the section. The default (if not specified) is to make the load address the same as the relocation address.

`contents` are made up of statements. These statements can:

- Assign a value to a symbol.
- Describe the placement of an output section, including which input sections are placed into it.

`segmentName` is the predefined memory segment into which you want to put the contents of the section. The two variants are:

<code>&gt;segmentName</code>	Places the section contents at the beginning of the memory segment <code>segmentName</code> .
<code>&gt;&gt;segmentName</code>	Appends the section contents to the memory segment <code>segmentName</code> .

## ELF Linker

### Linker Command File Keyword Listing

---

#### Example

```
SECTIONS {
    .text : {
        F_textSegmentStart = .;
        footpad.c (.text)
        . = ALIGN (0x10);
        padfoot.c (.text)
        F_textSegmentEnd = .;
    } > TEXT
    .data : { *(.data) } > DATA
    .bss  : { *(.bss) > BSS
             *(COMMON)
    }
}
```

---

## SIZEOF

The SIZEOF function returns the size of the given segment or section. The return value is the size in bytes.

### Prototype

`SIZEOF(sectionName | segmentName | symbol)`

---

#### NOTE

In order to use *segmentName* with this command, the *segmentName* must start with the period character even though *segmentNames* are not required to start with the period character by the linker, as is the case with *sectionName*.

---

---

## SIZEOFW

The SIZEOFW function returns the size of the given segment or section. The return value is the size in words.

### Prototype

```
SIZEOFW(sectionName | segmentName | symbol)
```

---

In order to use *segmentName* with this command, the *segmentName* must start with the period character even though *segmentNames* are not required to start with the period character by the linker, as is the case with *sectionName*.

---

---

## WRITEB

The WRITEB command inserts a byte of data at the current address of a section.

### Prototype

```
WRITEB (expression);
```

*expression* is any expression that returns a value 0x00 to 0xFF.

---

## WRITEH

The WRITEH command inserts two bytes of data at the current address of a section.

### Prototype

```
WRITEH (expression);
```

*expression* is any expression that returns a value 0x0000 to 0xFFFF.

## ELF Linker

*Linker Command File Keyword Listing*

---

### WRITEW

The WRITEW command inserts 4 bytes of data at the current address of a section.

#### Prototype

```
WRITEW (expression);
```

*expression* is any expression that returns a value 0x00000000 to 0xFFFFFFFF.

# Command-Line Tools

---

This chapter contains the following sections:

- Usage
- Response File
- Sample Build Script
- Arguments

## Usage

To call the command-line tools, use the following format:

**Table 13.1**    **Format**

Tools	File Names	Format
Compiler	mwcc56800e.exe	compiler-options [linker-options] file-list
Linker	mwld56800e.exe	linker-options file-list
Assembler	mwasm56800e.exe	assembler-options file-list

The compiler automatically calls the linker by default and any options from the linker is passed on by the compiler to the assembler. However, you may choose to only compile with the `-c` flag. In this case, the assembler will only assemble and will not call the linker.

Also, available are environment variables. These are used to provide path information for includes or libraries, and to specify which libraries are to be included. You can specify the variables listed in Table 13.2.

**Table 13.2 Environment Variables**

Tool	Library	Description
Compiler	MWC56800EIncludes	Similar to Access Paths panel; separate paths with ';' and prefix a path with '+' to specify a recursive path
Linker	MW56800ELibraries	Similar to MWC56800EIncludes
	MW56800ELibraryFiles	List of library names to link with project; separate with ';'.
Assembler	MWAsm56800EIncludes	(similar to MWC56800EIncludes)

These are the target-specific variables, and will only work with the DSP56800E tools. The generic variables **MWCIncludes**, **MWLibraries**, **MWLibraryFiles**, and **MWAsmIncludes** apply to all target tools on your system (such as Windows). If you only have the DSP56800E tools installed, then you may use the generic variables if you prefer.

## Response File

In addition to specifying commands in the argument list, you may also specify a “response file”. A response file’s filename begins with an ‘@’ (for example, @file), and the contents of the response file are commands to be inserted into the argument list. The response file supports standard UNIX-style comments. For example, the response file @file, contain the following:

```
# Response file @file
-o out.elf      # change output file name to 'out.elf'
-g             # generate debugging symbols
```

The above response file can used in a command such as:

**mwcc56800e** @file main.c

It would be the same as using the following command:

**mwcc56800e** -o out.elf -g main.c

---

## Sample Build Script

The following is a sample of a DOS batch (BAT) file. The sample demonstrates:

- Setting of the environmental variables.
- Using the compiler to compile and link a set of files.

---

```
REM *** set GUI compiler path ***
set COMPILER={path to compiler}

REM *** set includes path ***
set MWCIncludes=+%COMPILER%\M56800E Support
set MWLibraries=+%COMPILER%\M56800E Support
set MWLibraryFiles=Runtime 56800E.Lib;MSL C 56800E.lib

REM *** add CLT directory to PATH ***
set
PATH=%PATH%;%COMPILER%\DSP56800E_EABI_Tools\Command_Line_Tools\

REM *** compile options and files ***
set COPTIONS=-O3
set CFILELIST=file1.c file2.c
set LOPTIONS=-m FSTART_ -o output.elf -g
set LCF=linker.cmd

REM *** compile, assemble and link ***
mwcc56800e %COPTIONS% %CFILELIST%
mwasm56800e %AFILELIST%
mwld56800e %LOPTIONS% %LFILELIST% %LCF%
```

---

## Arguments

### General Command-Line Options

---

-----  
General Command-Line Options

All the options are passed to the linker unless otherwise noted.

Please see '-help usage' for details about the meaning of this help.

# Freescale Semiconductor, Inc.

## Command-Line Tools

### Arguments

---

```
-----
-help [keyword[,...]]      # global; for this tool;
                           #   display help
    usage                  #   show usage information
    [no]spaces              #   insert blank lines between options in
                           #   printout
    all                    #   show all standard options
    [no]normal              #   show only standard options
    [no]obsolete            #   show obsolete options
    [no]ignored             #   show ignored options
    [no]deprecated          #   show deprecated options
    [no]meaningless         #   show options meaningless for this
target
    [no]compatible          #   show compatibility options
    opt[ion]=name           #   show help for a given option; for 'name',
                           #   maximum length 63 chars
    search=keyword          #   show help for an option whose name or help
                           #   contains 'keyword' (case-sensitive); for
                           #   'keyword', maximum length 63 chars
    group=keyword           #   show help for groups whose names contain
                           #   'keyword' (case-sensitive); for 'keyword'
                           #   maximum length 63 chars
    tool=keyword[,...]     #   categorize groups of options by tool;
                           #   default
    all                     #   show all options available in this tool
    this                    #   show options executed by this tool
                           #   default
    other|skipped           #   show options passed to another tool
    both                    #   show options used in all tools
                           #
                           #
    -version                # global; for this tool;
                           #   show version, configuration, and build
date
    -timing                  # global; collect timing statistics
    -progress               # global; show progress and version
    -v[erbose]              # global; verbose information; cumulative;
                           #   implies -progress
    -search                 # global; search access paths for source files
                           #   specified on the command line; may specify
                           #   object code and libraries as well; this
                           #   option provides the IDE's 'access paths'
                           #   functionality
    -[no]wraplines          # global; word wrap messages; default
    -maxerrors max          # specify maximum number of errors to print, zero
```

---



---

```

                                # means no maximum; default is 0
-maxwarnings max      # specify maximum number of warnings to print,
                                # zero means no maximum; default is 0
-msgstyle keyword     # global; set error/warning message style
    mpw               # use MPW message style
    std               # use standard message style; default
    gcc               # use GCC-like message style
    IDE               # use CW IDE-like message style
    parseable         # use context-free machine-parseable message
                                # style
                                #
-[no]stderr            # global; use separate stderr and stdout streams;
                                # if using -nostderr, stderr goes to stdout

```

---

## Compiler

---

### ----- Preprocessing, Precompiling, and Input File Control Options -----

```

-c                     # global; compile only, do not link
-[no]codegen           # global; generate object code
-[no]convertpaths      # global; interpret #include filepaths specified
                                # for a foreign operating system; i.e.,
                                # <sys/stat.h> or <:sys:stat.h>; when enabled,
                                # '/' and ':' will separate directories and
                                # cannot be used in filenames (note: this is
                                # not a problem on Win32, since these
                                # characters are already disallowed in
                                # filenames; it is safe to leave the option
                                # 'on'); default
-cwd keyword           # specify #include searching semantics: before
                                # searching any access paths, the path
                                # specified by this option will be searched
    proj               # begin search in current working directory;
                                # default
    source              # begin search in directory of source file
    explicit            # no implicit directory; only search '-I' or
                                # '-ir' paths
    include             # begin search in directory of referencing
                                # file
                                #
-D+ | -d[efine         # cased; define symbol 'name' to 'value' if
    name[=value]       # specified, else '1'

```

---

# Freescale Semiconductor, Inc.

## Command-Line Tools Arguments

---

```
-[no]defaults      # global; passed to linker;
                   #   same as '-[no]stdinc'; default
-dis[assemble]     # global; passed to all tools;
                   #   disassemble files to stdout
-E                 # global; cased; preprocess source files
-EP                # global; cased; preprocess and strip out
#line              #   directives
-ext extension      # global; specify extension for generated object
                   #   files; with a leading period ('.'), appends
                   #   extension; without, replaces source file's
                   #   extension; for 'extension', maximum length 14
                   #   chars; default is none
-gccinc[ludes]      # global; adopt GCC #include semantics: add '-I'
                   #   paths to system list if '-I-' is not
                   #   specified, and search directory of
                   #   referencing file first for #includes (same as
                   #   '-cwd include')
-i- | -I-          # global; change target for '-I' access paths to
                   #   the system list; implies '-cwd explicit';
                   #   while compiling, user paths then system paths
                   #   are searched when using '#include "..."; only
                   #   system paths are searched with '#include
                   #   <...>'
-I+ | -i p          # global; cased; append access path to current
                   #   #include list(see '-gccincludes' and '-I-')
-ir path            # global; append a recursive access path to
                   #   current #include list
-[no]keepobj[ects]  # global; keep object files generated after
                   #   invoking linker; if disabled, intermediate
                   #   object files are temporary and deleted after
                   #   link stage; objects are always kept when
                   #   compiling
-M                 # global; cased; scan source files for
                   #   dependencies and emit Makefile, do not
                   #   generate object code
-MM                # global; cased; like -M, but do not list system
                   #   include files
-MD                # global; cased; like -M, but write dependency
                   #   map to a file and generate object code
-MMD               # global; cased; like -MD, but do not list system
                   #   include files
-make              # global; scan source files for dependencies and
                   #   emit Makefile, do not generate object
code -nofail        # continue working after errors in earlier files
```

---

```

-nolink                # global; compile only, do not link
-noprecompile          # do not precompile any files based on the
                        #   filename extension
-nosyspath             # global; treat #include <...> like #include
                        #   "..."; always search both user and system
                        #   path lists
-o file|dir            # specify output filename or directory for object
                        #   file(s) or text output, or output filename
                        #   for linker if called
-P                    # global; cased; preprocess and send output to
                        #   file; do not generate code
-precompile file|di    # generate precompiled header from source; write
                        #   header to 'file' if specified, or put header
                        #   in 'dir'; if argument is "", write header to
                        #   source-specified location; if neither is
                        #   defined, header filename is derived from
                        #   source filename; note: the driver can tell
                        #   whether to precompile a file based on its
                        #   extension; '-precompile file source' then is
                        #   the same as '-c -o file source'
-preprocess            # global; preprocess source files
-prefix file           # prefix text file or precompiled header onto all
                        #   source files
-S                    # global; cased; passed to all tools;
                        #   disassemble and send output to file
-[no]stdinc            # global; use standard system include paths
                        #   (specified by the environment variable
                        #   %MWCIncludes%); added after all system '-I'
                        #   paths; default
-U+ | -u[ndefine] name # cased; undefine symbol 'name'

```

---

## Front-End C/C++ Language Options

---

```

-ansi keyword          # specify ANSI conformance options, overriding
                        #   the given settings
    off                #   same as '-stdkeywords off', '-enum min', and
                        #   '-strict off'; default
    on|relaxed          #   same as '-stdkeywords on', '-enum min', and
                        #   '-strict on'
    strict              #   same as '-stdkeywords on', '-enum int', and
                        #   '-strict on'
                        #
-ARM on|off            # check code for ARM (Annotated C++ Reference
                        #   Manual) conformance; default is off

```

# Freescale Semiconductor, Inc.

## Command-Line Tools

### Arguments

---

```
-bool on|off          # enable C++ 'bool' type, 'true' and 'false'
                      # constants; default is off
-char keyword         # set sign of 'char'
  signed              # chars are signed; default
  unsigned            # chars are unsigned
                      #
-Cpp_exceptions on|off # passed to linker;
                      # enable or disable C++ exceptions; default is
                      # on
-dialect | -lang keyword # passed to linker;
                        # specify source language
  c                   # treat source as C always
  c++                 # treat source as C++ always
  ec++                # generate warnings for use of C++ features
                      # outside Embedded C++ subset (implies
                      # 'dialect cplus')
                      # 'dialect cplus')
-enum keyword         # specify word size for enumeration types
  min                 # use minimum sized enums; default
  int                 # use int-sized enums
                      #
-inline keyword[,...] # specify inline options
  on|smart            # turn on inlining for 'inline' functions;
                      # default
  none|off            # turn off inlining
  auto               # auto-inline small functions (without
                      # 'inline' explicitly specified)
  noauto             # do not auto-inline; default
  all                # turn on aggressive inlining: same as
                      # '-inline on, auto'
  deferred           # defer inlining until end of compilation
                      # unit; this allows inlining of functions in
                      # both directions
  level=n            # cased; inline functions up to 'n' levels
                      # deep; level 0 is the same as '-inline on';
                      # for 'n', range 0 - 8
                      #
-iso_templates on|off # enable ISO C++ template parser (note: this
                      # requires a different MSL C++ library);
                      # default is off
-[no]mapcr            # reverse mapping of '\n' and '\r' so that
                      # '\n'==13 and '\r'==10 (for Macintosh MPW
                      # compatibility)
-msextr keyword       # [dis]allow Microsoft VC++ extensions
  on                  # enable extensions: redefining macros,
```

---

---

```

#      allowing XXX::yyy syntax when declaring
#      method yyy of class XXX,
#      allowing extra commas,
#      ignoring casts to the same type,
#      treating function types with equivalent
#      parameter lists but different return types
#      as equal,
#      allowing pointer-to-integer conversions,
#      and various syntactical differences
off      #      disable extensions; default on non-x86
#      targets
#
-[no]multibyte[aware] # enable multi-byte character encodings for
#      source text, comments, and strings
-once      # prevent header files from being processed more
#      than once
-pragma      # define a pragma for the compiler such as
#      "#pragma ..."
-r[erequisite] # require prototypes
-relax_pointers # relax pointer type-checking rules
-RTTI on|off # select run-time typing information (for C++);
#      default is on
-som      # enable Apple's Direct-to-SOM implementation
-som_env_check # enables automatic SOM environment and new
#      allocation checking; implies -som
-stdkeywords on|off # allow only standard keywords; default is off
-str[ings] keyword[,...] # specify string constant options
[no]reuse # reuse strings; equivalent strings are the
#      same object; default
[no]pool # pool strings into a single data object
[no]readonly # make all string constants read-only
#
-strict on|off # specify ANSI strictness checking; default is
#      off
-trigraphs on|off # enable recognition of trigraphs; default is off
-wchar_t on|off # enable wchar_t as a built-in C++ type; default
#      is on

```

## ----- Optimizer Options

Note that all options besides '-opt  
off|on|all|space|speed|level=...' are  
for backwards compatibility; other optimization options may be  
superceded

## Command-Line Tools

### Arguments

---

by use of '-opt level=xxx'.

---

```

-O                                # same as '-O2'
-O+keyword[,...]                 # cased; control optimization; you may combine
                                # options as in '-O4,p'
    0                            # same as '-opt off'
    1                            # same as '-opt level=1'
    2                            # same as '-opt level=2'
    3                            # same as '-opt level=3'
    4                            # same as '-opt level=4'
    p                            # same as '-opt speed'
    s                            # same as '-opt space'
                                #
-opt keyword[,...]               # specify optimization options
    off|none                     # suppress all optimizations; default
    on                           # same as '-opt level=2'
    all|full                     # same as '-opt speed, level=4'
    [no]space                   # optimize for space
    [no]speed                   # optimize for speed
    l[level]=num                # set optimization level:
                                #     level 0: no optimizations
                                #
                                #     level 1: global register allocation,
                                #     peephole, dead code elimination
                                #
                                #     level 2: adds common subexpression
                                #     elimination and copy propagation
                                #
                                #     level 3: adds loop transformations,
                                #     strength reduction, loop-invariant code
                                #     motion
                                #
                                #     level 4: adds repeated common
                                #     subexpression elimination and
                                #     loop-invariant code motion
                                #
                                # ; for 'num', range 0 - 4; default is 0
    [no]cse                     # common subexpression elimination
    [no]commonsubs              #
    [no]deadcode                # removal of dead code
    [no]deadstore               # removal of dead assignments
    [no]lifetimes               # computation of variable lifetimes
    [no]loop[invariants]        # removal of loop invariants
    [no]prop[agation]           # propagation of constant and copy assignments
    [no]strength                # strength reduction; reducing multiplication
                                #     by an index variable into addition

```

---

---

[no]dead	#	same as '-opt [no]deadcode' and '-opt [no]deadstore'
display dump	#	display complete list of active optimizations
	#	
-----		
DSP M56800E CodeGen Options		
-----		
-DO keyword	#	for this tool;
	#	specify hardware DO loops
off	#	no hardware DO loops; default
nonested	#	hardware DO loops but no nested ones
nested	#	nested hardware DO loops
	#	
-padpipe	#	for this tool;
	#	pad pipeline for debugger
-ldata   -largedata	#	for this tool;
	#	data space not limited to 64K
-globalsInLowerMemory	#	for this tool;
	#	globals live in lower memory; implies '-large data model'
-sprog   -smallprog	#	for this tool;
	#	program space limited to 64K
-----		
Debugging Control Options		
-----		
-g	#	global; cased; generate debugging information;
	#	same as '-sym full'
-sym keyword[,...]	#	global; specify debugging options
off	#	do not generate debugging information;
	#	default
on	#	turn on debugging information
full[path]	#	store full paths to source files
	#	
-----		
C/C++ Warning Options		
-----		
-w[arn[ings]]	#	global; for this tool;
keyword[,...]	#	warning options
off	#	passed to all tools;
	#	turn off all warnings
on	#	passed to all tools;
	#	turn on most warnings

---

# Freescale Semiconductor, Inc.

## Command-Line Tools

### Arguments

---

[no]cmdline	#	passed to all tools;
	#	command-line driver/parser warnings
[no]err[or]	#	passed to all tools;
[no]iserr[or]	#	treat warnings as errors
all	#	turn on all warnings, require prototypes
[no]pragmas	#	illegal #pragmas
[no]illpragmas	#	
[no]empty[decl]	#	empty declarations
[no]possible	#	possible unwanted effects
[no]unwanted	#	
[no]unusedarg	#	unused arguments
[no]unusedvar	#	unused variables
[no]unused	#	same as -w [no]unusedarg, [no]unusedvar
[no]extracomma	#	extra commas
[no]comma	#	
[no]pedantic	#	pedantic error checking
[no]extended	#	
[no]hidevirtual	#	hidden virtual functions
[no]hidden[virtual]	#	
[no]implicit[conv]	#	implicit arithmetic conversions
[no]notinlined	#	'inline' functions not inlined
[no]largeargs	#	passing large arguments to unprototyped
	#	functions
[no]structclass	#	inconsistent use of 'class' and 'struct'
[no]padding	#	padding added between struct members
[no]notused	#	result of non-void-returning function not
	#	used
[no]unusedexpr	#	use of expressions as statements without
	#	side effects
[no]ptringconv	#	conversions from pointers to integers, and
	#	vice versa
display dump	#	display list of active warnings
	#	

---

## Linker

---

### Command-Line Linker Options

---

-dis[assemble]	#	global; disassemble object code and do not
	#	link; implies '-nostdlib'
-L+   -l path	#	global; cased; add library search path; default

---



---

```

# is to search current working directory and
# then system directories (see '-defaults');
# search paths have global scope over the
# command line and are searched in the order
# given
-lr path      # global; like '-l', but add recursive library
# search path
-l+file       # cased; add a library by searching access paths
# for file named lib<file>.<ext> where <ext> is
# a typical library extension; added before
# system libraries (see '-defaults')
-[no]defaults # global; same as -[no]stdlib; default
-nofail       # continue importing or disassembling after
# errors in earlier files
-[no]stdlib   # global; use system library access paths
# (specified by %MWLibraries%) and add system
# libraries (specified by
%MWLibraryFiles%);
# default
-S            # global; cased; disassemble and send output to
# file; do not link; implies '-nostdlib'

```

### ELF Linker Options

---

```

-[no]dead[strip] # enable dead-stripping of unused code; default
-force_active    # specify a list of symbols as undefined; useful
  symbol[,...]   # to force linking of static libraries
#
-keep[local] on|off # keep local symbols (such as relocations and
# output segment names) generated during link;
# default is on
-m[ain] symbol    # set main entry point for application or shared
# library; use '-main ""' to specify no entry
# point; for 'symbol', maximum length 63 chars;
# default is 'FSTART_'
-map [keyword[,...]] # generate link map file
  closure          # calculate symbol closures
  unused           # list unused symbols
#
-sortbyaddr       # sort S-records by address; implies '-srec'
-srec             # generate an S-record file; ignored when
# generating static libraries
-sreceol keyword  # set end-of-line separator for S-record file;
# implies '-srec'

```

# Freescale Semiconductor, Inc.

## Command-Line Tools

### Arguments

---

```
mac          # Macintosh ('\r')
dos          # DOS ('\r\n'); default
unix        # Unix ('\n')
#
-sreclength length # specify length of S-records (should be a
# multiple of 4); implies '-srec'; for
# 'length', range 8 - 252; default is 64
-usebyteaddr # use byte address in S-record file; implies
# '-srec'
-o file      # specify output filename
```

---

DSP M56800E Project Options

---

```
-application # global; generate an application; default
-library     # global; generate a static library
```

---

DSP M56800E CodeGen Options

---

```
-ldata | -largedata # data space not limited to 64K
```

---

Linker C/C++ Support Options

---

```
-Cpp_exceptions on|off # enable or disable C++ exceptions; default is
on
-dialect | -lang keyword # specify source language
c          # treat source as C++ unless its extension is
#          # '.c', '.h', or '.pch'; default
c++        # treat source as C++ always
#
```

---

Debugging Control Options

---

```
-g          # global; cased; generate debugging information;
#          # same as '-sym full'
-sym keyword[,...] # global; specify debugging options
off          # do not generate debugging information;
#          # default
on           # turn on debugging information
```

---

---

```

    full[path]          #   store full paths to source files
                        #

```

---

Warning Options

---

```

-w[arn[ings]]          #   global; warning options
keyword[,...]          #
    off                #   turn off all warnings
    on                 #   turn on all warnings
[no]cmdline            #   command-line parser warnings
[no]err[or] |          #   treat warnings as errors
    [no]iserr[or]      #
display|dump           #   display list of active warnings
                        #

```

---

ELF Disassembler Options

---

```

-show keyword[,...]    #   specify disassembly options
    only|none          #   as in '-show none' or, e.g.,
                        #   '-show only,code,data'
    all                #   show everything; default
[no]code | [no]text    #   show disassembly of code sections; default
[no]comments           #   show comment field in code; implies '-show
                        #   code'; default
[no]extended           #   show extended mnemonics; implies '-show
                        #   code'; default
[no]data               #   show data; with '-show verbose', show hex
                        #   dumps of sections; default
[no]debug | [no]sym     #   show symbolics information; default
[no]exceptions         #   show exception tables; implies '-show data';
                        #   default
[no]headers            #   show ELF headers; default
[no]hex                #   show addresses and opcodes in code
                        #   disassembly; implies '-show code'; default
[no]names              #   show symbol table; default
[no]relocs             #   show resolved relocations in code and
                        #   relocation tables; default
[no]source             #   show source in disassembly; implies '-show
                        #   code'; with '-show verbose', displays
                        #   entire source file in output, else shows
                        #   only four lines around each function;
                        #   default
[no]xtables            #   show exception tables; default

```

---

## Command-Line Tools

### Arguments

---

[no]verbose	#	show verbose information, including hex dump
	#	of program segments in applications;
	#	default
	#	

---

## Assembler

### Assembler Control Options

---

-[no]case	#	identifiers are case-sensitive; default
-[no]debug	#	generate debug information
-[no]macro_expand	#	expand macro in listin output
-[no]assert_nop	#	add nop to resolve pipeline dependency; default
-[no]warn_nop	#	emit warning when there is a pipeline dependency
-[no]warn_stall	#	emit warning when there is a hardware stall
-[no]legacy	#	allow legacy DSP56800 instructions (imply data/prog 16)
-[no]debug_workaround	#	Pad nop workaround debuggin issue in some implementation; default
-data keyword	#	data memory compatibility
16	#	16 bit; default
24	#	24 bit
	#	
-prog keyword	#	program memory compatibility
16	#	16 bit; default
19	#	19 bit
21	#	21 bit
	#	

---

# Libraries and Runtime Code

---

You can use a variety of libraries with the CodeWarrior™ IDE. The libraries include ANSI-standard libraries for C, runtime libraries, and other codes. This chapter explains how to use these libraries for DSP56800E development.

With respect to the Metrowerks Standard Library (MSL) for C, this chapter is an extension of the *MSL C Reference*. Consult that manual for general details on the standard libraries and their functions.

This chapter contains the following sections:

- MSL for DSP56800E
- Runtime Initialization
- EOnCE Library

## MSL for DSP56800E

This section explains the Metrowerks Standard Library (MSL) that has been modified for use with DSP56800E.

### Using MSL for DSP56800E

CodeWarrior Development Studio for Freescale 56800/E Hybrid Controllers includes a version of the Metrowerks Standard Library (MSL). MSL is a complete C library for use in embedded projects. All of the sources necessary to build MSL are included in CodeWarrior Development Studio for Freescale 56800/E Hybrid Controllers, along with the project files for different configurations of MSL. If you already have a version of the CodeWarrior IDE installed on your computer, the CodeWarrior installer adds the new files needed for building versions of MSL for DSP56800E.

The project directory for the DSP56800E MSL is:

# Freescale Semiconductor, Inc.

## Libraries and Runtime Code

*MSL for DSP56800E*

---

```
CodeWarrior\M56800E Support\msl\MSL_C\DSP_56800E\projects\MSL C
56800E.mcp
```

---

Do not modify any of the source files included with MSL. If you need to make changes based on your memory configuration, make changes to the runtime libraries.

Ensure that you include one or more of the header files located in the following directory:

---

```
CodeWarrior\M56800E Support\msl\MSL_C\DSP_56800E\inc
```

---

When you add the relative-to-compiler path to your project, the appropriate MSL and runtime files will be found by your project. If you create your project from Stationery, the new project will have the proper support access path.

## Console and File I/O

DSP56800E Support provides standard C calls for I/O functionality with full ANSI/ISO standard I/O support with host machine console and file I/O for debugging sessions (Host I/O) through the JTAG port or HSST in addition to such standard C calls such as memory functions malloc() and free().

A minimal "thin" printf via "console\_write" and "fflush\_console" is provided in addition to standard I/O.

See the *MSL C Reference* manual (Metrowerks Standard Library).

## Library Configurations

There are Large Data Model and Small Data Model versions of all libraries. (Small Program Model default is off for all library and Stationery targets.)

Metrowerks Standard Library (MSL) provides standard C library support.

The Runtime libraries provide the target-specific low-level functions below the high-level MSL functions. There are two types of Runtime libraries:

- JTAG-based Host I/O
- HSST-based Host I/O.

For each project requiring standard C library support, a matched pair of MSL and Runtime libraries are required (SDM or LDM pairs).

The HSST library is added to HSST client-to-client DSP56800E targets. For more information see "High-Speed Simultaneous Transfer".

---

<b>NOTE</b>	DSP56800E stationery creates new projects with LDM and SDM targets and the appropriate libraries.
-------------	---

---

Below is a list of the DSP56800E libraries:

- Metrowerks Standard Libraries (MSL)
  - MSL C 56800E.lib  
Standard C library support for Small Data Model.
  - MSL C 56800E lmm.lib  
Standard C library support for Large Data Model.
- Runtime Libraries
  - runtime 56800E.lib  
Low-level functions for MSL support for Small Data Model with Host I/O via JTAG port.
  - runtime 56800E lmm.lib  
Low-level functions for MSL support for Large Data Model with Host I/O via JTAG port.
  - runtime\_hsst\_56800E.lib  
Low-level functions for MSL support for Small Data Model with Host I/O via HSST.
  - runtime\_hsst\_56800E\_lmm.lib  
Low-level functions for MSL support for Large Data Model with Host I/O via HSST.
- HSST Libraries

There are debug and release targets for SDM and LDM. The release targets have maximum optimization settings and debug info turned off. For more information see “High-Speed Simultaneous Transfer”.

  - hsst\_56800E.lib  
DSP 56800E HSST client functions for Small Data Model.
  - hsst\_56800E\_lmm.lib  
DSP56800E HSST client functions for Large Data Model.

## Host File Location

Files are created with `fopen` on the host machine as shown in Table 14.1.

**Table 14.1 Host File Creation Location**

fopen Filename Parameter	Host Creation Location
filename with no path	target project file folder
full path	location of full path

## Allocating Stacks and Heaps for the DSP56800E

Stationery linker command files (LCF) define heap, stack, and bss locations. LCFs are specific to each target board. When you use M56800E stationery to create a new project, CodeWarrior automatically adds the LCF to the new project.

See “ELF Linker,” for general LCF information. See each specific target LCF in Stationery for specific LCF information.

See Table 14.2 for the variables defined in each Stationery LCF.

**Table 14.2 LCF Variables and Address**

Variables	Address
_stack_addr	the start address of the stack
_heap_size	the size of the heap
_heap_addr	the start address of the heap
_heap_end	the end address of the heap
_bss_start	start address of memory reserved for uninitialized variables
_bss_end	end address of bss

To change the locations of these default values, modify the linker command file in your DSP56800E project.

---

**NOTE**                      Ensure that the stack and heap memories reside in data memory.

---



---

## Definitions

### Stack

The stack is a last-in-first-out (LIFO) data structure. Items are pushed on the stack and popped off the stack. The most recently added item is on top of the stack. Previously added items are under the top, the oldest item at the bottom. The "top" of the stack may be in low memory or high memory, depending on stack design and use. M56800E uses a 16-bit-wide stack.

### Heap

Heap is an area of memory reserved for temporary dynamic memory allocation and access. MSL uses this space to provide heap operations such as malloc. M56800E does not have an operating system (OS), but MSL effectively synthesizes some OS services such as heap operations.

### BSS

BSS is the memory space reserved for uninitialized data. The compiler will put all uninitialized data here. If the **Zero initialized globals live in data** checkbox in the M56800E Processor Panel is checked, the globals that are initialized to zero reside in the `.data` section instead of the `.bss` section. The stationery init code zeroes this area at startup. See the M56852 init (startup) code in this chapter for general information and the stationery init code files for specific target implementation details.

---

<b>NOTE</b>	Instead of accessing the original Stationery files themselves (in the Stationery folder), create a new project using Stationery which will make copies of the specific target board files such as the LCF.
-------------	--

---

## Runtime Initialization

The default init function is the bootstrap or glue code that sets up the DSP56800E environment before your code executes. This function is in the init file for each board-specific stationery project. The routines defined in the init file performs other tasks such as clearing the hardware stack, creating an interrupt table, and retrieving the stack start and exception handler addresses.

The final task performed by the init function is to call the `main()` function.

# Freescale Semiconductor, Inc.

## Libraries and Runtime Code

### Runtime Initialization

The starting point for a program is set in the **Entry Point** field in the **M56800E Linker** settings panel.

The project for the DSP56800E runtime is:

CodeWarrior\M56800E Support\runtime\_56800E\projects\Runtime\_56800E.mcp

**Table 14.3 Library Names and Locations**

Library Name	Location
Large Memory Model Runtime_56800E_lmm.lib	CodeWarrior\M56800E Support\runtime_56800E\lib
Small Memory Model Runtime_56800E.Lib	CodeWarrior\M56800E Support\runtime_56800E\lib

When creating a project from R1.1 or later Stationery, the associated init code is specific to the DSP56800E board. See the startup folder in the new project folder for the init code.

### Listing 14.1 Sample Initialization File (DSP56852EVM)

```
#
; -----
;
; 56852_init.asm
; Metrowerks, a Freescale Company
; sample
; description: main entry point to C code.
;             setup runtime for C and call main
;
; -----

;=====
; OMR mode bits
;=====
NL_MODE      EQU      $8000
CM_MODE      EQU      $0100
XP_MODE      EQU      $0080
R_MODE       EQU      $0020
SA_MODE      EQU      $0010
```

```
section rtlib

XREF F_stack_addr
org p:

GLOBAL Finit_M56852_

SUBROUTINE "Finit_M56852_",Finit_M56852_,Finit_M56852END-
Finit_M56852_

Finit_M56852_:

;
; setup the OMr with the values required by C
;
    bfset    #NL_MODE,omr        ; ensure NL=1 (enables nsted DO loops)
    nop
    nop
    bfclr    #(CM_MODE|XP_MODE|R_MODE|SA_MODE),omr ; ensure CM=0
(optional for C)
                ; ensure XP=0 to enable harvard architecture
                ; ensure R=0 (required for C)
                ; ensure SA=0 (required for C)

; Setup the m01 register for linear addressing
    move.w   #-1,x0
    moveu.w  x0,m01              ; Set the m register to linear addressing

    moveu.w  hws,la              ; Clear the hardware stack
    moveu.w  hws,la
    nop
    nop

CALLMAIN:                                ; Initialize compiler environment

;Initialize the Stack
    move.l   #>>>F_lstack_addr,r0
    bftsth  #$0001,r0
    bcc  noinc
    adda    #1,r0
```

# Freescale Semiconductor, Inc.

## Libraries and Runtime Code

### EOnCE Library

---

```
noinc:
    tfra    r0,sp                ; set stack pointer too
    move.w  #0,r1
    nop
    move.w  r1,x:(sp)
    adda    #1,sp

    jsr     F__init_sections

; Call main()
    move.w  #0,y0                ; Pass parameters to main()
    move.w  #0,R2
    move.w  #0,R3

    jsr     Fmain                ; Call the Users program
;
; The fflush calls were removed because they added code
; growth in cases where the user is not using any debugger IO.
; Users should now make these calls at the end of main if they use
; debugger IO
;
;    move.w  #0,r2
;    jsr     Ffflush              ; Flush File IO
;    jsr     Ffflush_console     ; Flush Console IO

;    end of program; halt CPU
    debuglt
    rts
Finit_M56852END:

    endsec
```

---

## EOnCE Library

The EOnCE (Enhanced On Chip Emulator) library provides functions, which allows your program to control the EOnCE. The library lets you set and clear triggers for breakpoints, watchpoints, program traces, and counters. With several option enumerations, the library greatly simplifies using the EOnCE from within the core, and thus eliminates the need for a DSP56800E User Manual. The library and the debugger are coordinated so that the debugger does not overwrite a trigger set by the library, and vice versa.

To use the EOnCE library, you must include it in your project. The name of the file is `eonce 56800E lmm.lib` and it is located at:

`CodeWarrior\M56800ESupport\eonce\lib`

The **Large Data Model** option must be enabled in the **M56800E Processor** preference panel. Any source file that contains code that calls any of the EOnCE Library functions must `#include eonceLib.h`. This header file is located at:

`CodeWarrior\M56800E Support\eonce\include`

The library functions are listed below:

- `_eonce_Initialize`
- `_eonce_SetTrigger`
- `_eonce_SetCounterTrigger`
- `_eonce_ClearTrigger`
- `_eonce_GetCounters`
- `_eonce_GetCounterStatus`
- `_eonce_SetupTraceBuffer`
- `_eonce_GetTraceBuffer`
- `_eonce_ClearTraceBuffer`
- `_eonce_StartTraceBuffer`
- `_eonce_HaltTraceBuffer`
- `_eonce_EnableDEBUGEV`
- `_eonce_EnableLimitTrigger`

The sub-section “Definitions” on page 339 defines:

- Return Codes
- Normal Trigger Modes
- Counter Trigger Modes
- Data Selection Modes
- Counter Function Modes
- Normal Unit Action Options
- Counter Unit Action Options
- Accumulating Trigger Options
- Miscellaneous Trigger Options

## Libraries and Runtime Code

### EOnCE Library

---

- Trace Buffer Capture Options
  - Trace Buffer Full Options
  - Miscellaneous Trace Buffer Option
- 

## **`_eonce_initialize`**

Initializes the library by setting the necessary variables.

### **Prototype**

```
void _eonce_initialize( unsigned long baseAddr, unsigned int  
                      units )
```

### **Parameters**

`baseAddr` unsigned long

Specifies the location in X: memory where the EOnCE registers are located.

`units` unsigned int

Specifies the number of EOnCE breakpoint units available.

### **Remarks**

This function must be called before any other library function is called. Its parameters are dependent on the processor being used. Instead of calling this function directly, one of the defined macros can be called in its place. These include **`_eonce_initialize56838E()`**, **`_eonce_initialize56852E()`**, and **`_eonce_initialize56858E()`**. These macros call **`_eonce_initialize`** with the correct parameters for the 56838, 56852, and 56858, respectively.

### **Returns**

Nothing.

---

## **`_eonce_SetTrigger`**

Sets a trigger condition used to halt the processor, cause an interrupt, or start and stop the trace buffer. This function does not set triggers for special counting functions.

### **Prototype**

```
int _eonce_SetTrigger( unsigned int unit, unsigned long options,  
    unsigned long value1, unsigned long value2, unsigned long  
    mask, unsigned int counter )
```

### **Parameters**

`unit` unsigned int

Specifies which breakpoint unit to use.

`options` unsigned long

Describes the behavior of the trigger. For more information on the identifiers for this parameter, please see the sub-section “Definitions” on page 325.

`value1` unsigned long

Specifies the address or data value to compare as defined by the options parameter.

`value2` unsigned long

Specifies the address or data value to compare as defined by the options parameter.

`mask` unsigned long

Specifies which bits of value2 to compare.

`counter` unsigned int

Specifies the number of successful comparison matches to count before completing trigger sequence as defined by the options parameter

### **Remarks**

This function sets all triggers, except those used to define the special counting function behavior. Carefully read the list of defined identifiers that can be OR’ed into the options parameter.

## Returns

Error code as defined in the sub-section “Definitions” on page 339.

---

## **`_eonce_SetCounterTrigger`**

Sets a trigger condition used for special counting functions.

## Prototype

```
int _eonce_SetCounterTrigger( unsigned int unit, unsigned long
    options, unsigned long value1, unsigned long value2,
    unsigned long mask, unsigned int counter, unsigned long
    counter2 )
```

## Parameters

`unit` unsigned int

Specifies which breakpoint unit to use.

`options` unsigned long

Describes the behavior of the trigger. For more information on the identifiers for this parameter, please see the sub-section “Definitions” on page 325.

`value1` unsigned long

Specifies the address or data value to compare as defined by the options parameter.

`value2` unsigned long

Specifies the address or data value to compare as defined by the options parameter.

`mask` unsigned long

Specifies which bit of value2 to compare.

`counter` unsigned int

Specifies the value used to pre-load the counter, which proceeds backward when `EXTEND_COUNTER` is OR'ed into the options parameter. `counter` contains the least significant 16-bits.



---

`counter2` unsigned long

Specifies the value used to pre-load the counter, which proceeds backward. When `EXTEND_COUNTER` is OR'ed into the options parameter, `counter2` contains the most significant 24-bits. However, when `EXTEND_COUNTER` is not OR'ed `counter2` should be set to 0.

### Remarks

This function is used to set special counting function triggers. The special counting options are defined in the sub-section "Definitions" on page 339. Carefully read the list of defined identifiers that can be OR'ed into the options parameter.

### Returns

Error code as defined in the sub-section "Definitions" on page 339.

---

## **`_eonce_ClearTrigger`**

Clears a previously set trigger.

### Prototype

```
int _eonce_ClearTrigger( unsigned int unit )
```

### Parameters

`unit` unsigned int

Specifies which breakpoint unit to use.

### Remarks

This function clears a trigger set with the `_eonce_SetTrigger` or `_eonce_SetCounterTrigger` functions.

### Returns

Error code as defined in the sub-section "Definitions" on page 339.

---

## **\_eonce\_GetCounters**

Retrieves the values in the two counter registers.

### **Prototype**

```
int _eonce_GetCounters( unsigned int unit, unsigned int
    *counter, unsigned long *counter2 )
```

### **Parameters**

unit unsigned int

Specifies which breakpoint unit to use.

counter unsigned int \*

Holds the value of the counter, or the least significant 16-bits, if the counter has been extended to 40-bits.

counter2 unsigned long \*

Holds the most significant 24-bits if the counter has been extended to 40-bits. This parameter must be a valid pointer even if the counter has not been extended.

### **Remarks**

This function retrieves the value of the counter of the specified breakpoint unit. This function is most useful when using the special counting function of the breakpoint, but can also be used to retrieve the trigger occurrence counter.

### **Returns**

Error code as defined in the sub-section “Definitions” on page 339.

---

## **\_eonce\_GetCounterStatus**

Retrieves the status of the breakpoint counter.

### **Prototype**

```
int _eonce_GetCounterStatus( char *counterIsZero, char
```

`*counterIsStopped )`

## Parameters

`counterIsZero char *`

Returns a 1 if the breakpoint counter has reached zero.

`counterIsStopped char *`

Returns a 1 if the breakpoint counter has been stopped by a Counter Stop Trigger.

## Remarks

This function returns the state of the breakpoint counter when using the special counting function.

## Returns

Error code as defined in the sub-section “Definitions” on page 339.

---

## **`_eonce_SetupTraceBuffer`**

Configures the behavior of the trace buffer.

## Prototype

`int _eonce_SetupTraceBuffer( unsigned int options )`

## Parameters

`options unsigned int`

Describes the behavior of the trace buffer. Please see the section Definitions for more information on the identifiers for this parameter.

## Remarks

Sets the behavior of the trace buffer. Triggers can also be set to start and stop trace buffer capture using the **`_eonce_SetTrigger`** function.

## Returns

Error code as defined in the sub-section “Definitions” on page 339.

---

## **\_eonce\_GetTraceBuffer**

Retrieves the contents of the trace buffer.

### **Prototype**

```
int _eonce_GetTraceBuffer( unsigned int *count, unsigned long
                          *buffer )
```

### **Parameters**

count unsigned int \*

Passes in the size of the buffer; if 0 is passed in, the contents of the trace buffer are not retrieved, instead the number of entries in the trace buffer are returned in count.

buffer unsigned long \*

Points to an array in which the contents of the trace buffer are returned starting with the oldest entry.

### **Remarks**

This function retrieves the addresses contained in the trace buffer. The addresses represent the program execution point when certain change-of-flow events occur. The trace buffer behavior, including capture events, can be configured using **\_eonce\_SetupTraceBuffer**.

### **Returns**

Error code as defined in the sub-section “Definitions” on page 339.

---

## **\_eonce\_ClearTraceBuffer**

Clears the contents of the trace buffer.

### **Prototype**

```
int _eonce_ClearTraceBuffer( )
```

---

## Parameters

None.

## Remarks

This function clears the trace buffer and is useful when you want a fresh set of data. It is necessary to resume capturing when the trace buffer is full and configured to stop capturing.

## Returns

Error code as defined in the sub-section “Definitions” on page 339.

---

## **`_eonce_StartTraceBuffer`**

Resumes trace buffer capturing.

## Prototype

```
int _eonce_StartTraceBuffer( )
```

## Parameters

None.

## Remarks

This function causes the trace buffer to immediately start capturing.

## Returns

Error code as defined in the sub-section “Definitions” on page 339.

---

## **`_eonce_HaltTraceBuffer`**

Halts trace buffer capturing.

## Prototype

```
int _eonce_HaltTraceBuffer( )
```

## **Parameters**

None.

## **Remarks**

Causes the trace buffer to immediately stop capturing.

## **Returns**

Error code as defined in the sub-section “Definitions” on page 339.

---

## **\_eonce\_EnableDEBUGEV**

Allows or disallows a DEBUGEV instruction to cause a core event in breakpoint unit 0.

## **Prototype**

```
int _eonce_EnableDEBUGEV( char enable )
```

## **Parameters**

`enable`char

If a non-zero value, allows the DEBUGEV instruction to cause a core event. If a zero value, prevents the DEBUGEV instruction from causing a core event.

## **Remarks**

This function configures the behavior for the DEBUGEV instructions. For a core event to occur, breakpoint unit 0 must be activated by setting a trigger using the **`_eonce_SetTrigger`** or **`_eonce_SetCounterTrigger`** functions.

## **Returns**

Error code as defined in the sub-section “Definitions” on page 339.

---

## **\_eonce\_EnableLimitTrigger**

Allows or disallows a limit trigger to cause a core event in breakpoint unit 0.

## Prototype

```
int _eonce_EnableLimitTrigger( char enable )
```

## Parameters

enablechar

If a non-zero value, allows this instruction to cause a core event. If a zero value, prevents this instruction from causing a core event.

## Remarks

This function configures the behavior for overflow and saturation conditions in the processor core. For a core event to occur, breakpoint unit 0 must be activated by setting a trigger using the **\_eonce\_SetTrigger** or **\_eonce\_SetCounterTrigger** functions.

## Returns

Error code as defined in the sub-section “Definitions” on page 339.

## Definitions

This sub-section defines:

- Return Codes
- Normal Trigger Modes
- Counter Trigger Modes
- Data Selection Modes
- Counter Function Modes
- Normal Unit Action Options
- Counter Unit Action Options
- Accumulating Trigger Options
- Miscellaneous Trigger Options
- Trace Buffer Capture Options
- Trace Buffer Full Options
- Miscellaneous Trace Buffer Option

## Return Codes

Every function except **\_eonce\_Initialize** returns one of the error codes in Table 14.4.

**Table 14.4 Error Codes**

Error Code	Description
EONCE_ERR_NONE	No error.
EONCE_ERR_NOT_INITIALIZED	The <b>_eonce_Initialize</b> function has not been called before the current function.
EONCE_ERR_UNIT_OUT_OF_RANGE	The unit parameter is greater than or equal to the number of units specified in <b>_eonce_Initialize</b> .
EONCE_ERR_LOCKED_OUT	The core cannot access the EOnCE registers because the debugger has locked out the core. This occurs when a trigger has been set using the EOnCE GUI panels or through an IDE breakpoint or watchpoint.

## Normal Trigger Modes

One of the defined identifiers listed in Listing 14.2 must be OR'ed into the options parameter of the **\_eonce\_SetTrigger** function. A key for the defined identifiers listed in Listing 14.2 is given in Table 14.5.

**Listing 14.2 Normal Trigger Modes**

```

B1PA_N
B1PR_N
B1PW_N
B2PF_N
B1XA_OR_B2PF_N
B1XA_N_OR_B2PF
B1PF_OR_B2PF_N
B1PA_OR_B2PF_N
B1PA_N_OR_B2PF
B1PF_OR_N_B2PF
B1PA_OR_N_B2PF
B1XR_AND_N_B2DR
B1XW_AND_N_B2DW
B1XA_AND_N_B2DRW
B1PF_N_THEN_B2PF
B2PF_THEN_B1PF_N
B1PA_N_THEN_B2PF

```



B1PA\_THEN\_B2PF\_N  
B2PF\_N\_THEN\_B1PA  
B2PF\_THEN\_B1PA\_N  
B1XA\_N\_THEN\_B2PF  
B1XA\_THEN\_B2PF\_N  
B2PF\_N\_THEN\_B1XA  
B2PF\_THEN\_B1XA\_N  
B1XW\_N\_THEN\_B2PF  
B1XW\_THEN\_B2PF\_N  
B2PF\_N\_THEN\_B1XW  
B2PF\_THEN\_B1XW\_N  
B1XR\_N\_THEN\_B2PF  
B1XR\_THEN\_B2PF\_N  
B2PF\_N\_THEN\_B1XR  
B2PF\_THEN\_B1XR\_N  
B1PF\_STB\_B2PF\_HTB  
B1PA\_STB\_B2PF\_HTB  
B2PF\_STB\_B1PA\_HTB

Defined Identifier Key for Normal Trigger Modes

**Table 14.5 Defined Identifier Key: Normal Trigger Modes**

Identifier Fragments	Description
B1	breakpoint 1; value set in value1
B2	breakpoint 2; value set in value2
P	p-memory address; this is followed by a type of access
X	x-memory address; this is followed by a type of access
D	value being read from or written to x-memory
A	memory access
R	memory read
W	memory write
F	memory fetch; only follows a P
OR	links two sub-triggers by a logical or
AND	links two sub-triggers by a logical and
THEN	creates a sequence; first sub-trigger must occur, then second sub-trigger must occur to complete the trigger

**Table 14.5 Defined Identifier Key: Normal Trigger Modes (*continued*)**

Identifier Fragments	Description
N	the sub-trigger it follows must occur N times as set in the count parameter; if N follows an operation, then the combination of the sub-triggers must occur N times; (count - 1) will be written to the BCNTR register
STB	sub-trigger starts the trace buffer
HTB	sub-trigger halts the trace buffer

## Counter Trigger Modes

The following triggers generate a Counter Stop Trigger. The exceptions are the modes that generate both start and stop triggers.

The defined identifiers listed in Listing 14.3 must be OR'ed into the options parameter of the **\_eonce\_SetCounterTrigger** function. A key for the defined identifiers listed in Listing 14.3 is given in Table 14.6

**Listing 14.3 Counter Trigger Modes**

```

B1PA
B1PR
B1PW
B2PF
B1XA_OR_B2PF
B1PF_OR_B2PF
B1PA_OR_B2PF

B1XR_AND_B2DR
B1XW_AND_B2DW
B1XA_AND_B2DRW
B1PF_THEN_B2PF
B1PA_THEN_B2PF
B2PF_THEN_B1PA
B1XA_THEN_B2PF
B2PF_THEN_B1XA
B1XW_THEN_B2PF
B2PF_THEN_B1XW
B1XR_THEN_B2PF
B2PF_THEN_B1XR
B1PF_SC_B2PF_HC

```

B1PA\_SC\_B2PF\_HC  
B2PF\_SC\_B1PA\_HC

**Table 14.6 Defined Identifier Key: Counter Trigger Modes**

Identifier Fragments	Description
B1	breakpoint 1; value set in value1
B2	breakpoint 2; value set in value2
P	p-memory address; this is followed by a type of access
X	x-memory address; this is followed by a type of access.
D	value being read from or written to x-memory
A	memory access
R	memory read
W	memory write
F	memory fetch; only follows a P
OR	links two sub-triggers by a logical or
AND	links two sub-triggers by a logical and
THEN	creates a sequence; first sub-trigger must occur, then second sub-trigger must occur to complete the trigger
SC	sub-trigger starts the counter
HC	sub-trigger halts the counter

## Data Selection Modes

If the trigger mode being set includes a data value compare (contains B2D from the list Normal Trigger Modes or Counter Trigger Modes), then one of the defined identifiers in Table 14.7 must be OR'ed into the options parameter of the **\_eonce\_SetTrigger** or **\_eonce\_SetCounterTrigger** function. If not, then do not OR in any of these identifiers.

**Table 14.7 Data Selection Modes**

Defined Identifiers	Description
B2D_BYTE	makes a comparison when the data being moved is of byte-length

**Table 14.7 Data Selection Modes**

Defined Identifiers	Description
B2D_WORD	makes a comparison when the data being moved is of word-length
B2D_LONG	makes a comparison when the data being moved is of long-length

## Counter Function Modes

One of the defined identifiers in Table 14.8 must be OR'ed into the options parameter of the `_eonce_SetCounterTrigger` function.

**Table 14.8 Counter Function Modes**

Defined Identifiers	Description
PCLK_CLOCK_CYCLES	count pclk cycles
CLK_CLOCK_CYCLES	count clk cycles
INSTRUCTIONS_EXECUTED	count instructions executed
TRACE_BUFFER_WRITES	count writes to the trace buffer
COUNTER_START_TRIGGERS	count Counter Start Triggers
PCLK_CLOCK_CYCLES	count pclk cycles

## Normal Unit Action Options

This list of options describes the action taken when a non-counter trigger is generated. One of the defined identifiers in Table 14.9 must be OR'ed into the options parameter of the `_eonce_SetTrigger` function.

**Table 14.9 Normal Unit Actions Options Mode**

Defined Identifiers	Description
UNIT_ACTION	enters debug mode is unit 0, else passes signal on to next unit
INTERRUPT_CORE	interrupts to vector set for this unit
HALT_TRACE_BUFFER	trace buffer capture is halted
START_TRACE_BUFFER	trace buffer capture is started
UNIT_ACTION	enters debug mode is unit 0, else passes signal on to next unit

## Counter Unit Action Options

This list of options describes the action taken when a counter trigger is generated. One of the defined identifiers in Table 14.10 must be OR'ed into the options parameter of the **\_eonce\_SetCounterTrigger** function. Identifiers that include **ZERO\_BEFORE\_TRIGGER** only perform the action when the counter counts down to zero before the Counter Stop Trigger occurs. Identifiers that include **TRIGGER\_BEFORE\_ZERO** only perform the action when the Counter Stop Trigger occurs before the counter counts down to zero.

**Table 14.10 Counter Unit Actions Options Mode**

Defined Identifiers	Description
NO_ACTION	counter status bits still get set
UNIT_ACTION_ZERO_BEFORE_TRIGGER	enters debug mode is unit 0, else passes signal on to next unit
INTERRUPT_CORE_ZERO_BEFORE_TRIGGER	interrupts to vector set for this unit
UNIT_ACTION_TRIGGER_BEFORE_ZERO	enters debug mode is unit 0, else passes signal on to next unit
INTERRUPT_CORE_TRIGGER_BEFORE_ZERO	interrupts to vector set for this unit

## Accumulating Trigger Options

One of the defined identifiers in Table 14.11 must be OR'ed into the options parameter of the **\_eonce\_SetTrigger** function when breakpoint unit 0 is being configured.

**Table 14.11 Accumulating Trigger Options Mode with Breakpoint Unit 0**

Defined Identifiers	Description
PREV_UNIT_OR_THIS_TRIGGER_OR_CORE_EVENT	a trigger is generated if the previous breakpoint unit passes in a trigger signal or this breakpoint unit creates a trigger signal or if a core event occurs
PREV_UNIT_THEN_THIS_TRIGGER_OR_CORE_EVENT	a trigger is generated if the previous breakpoint unit passes in a trigger signal followed by either this breakpoint unit creating a trigger signal or a core event occurring

**Table 14.11 Accumulating Trigger Options Mode with Breakpoint Unit 0**

Defined Identifiers	Description
THIS_TRIGGER_THEN_CORE_EVENT	a trigger is generated if this breakpoint unit creates a trigger signal followed by a core event occurring
PREV_UNIT_THEN_THIS_TRIGGER_THEN_CORE_EVENT	a trigger is generated if the previous breakpoint unit passes in a trigger signal followed by this breakpoint unit creating a trigger signal followed by a core event occurring

One of the defined identifiers in Table 14.12 must be OR'ed into the options parameter of the **\_once\_SetTrigger** function when a breakpoint unit other than unit 0 is being configured.

**Table 14.12 Accumulating Trigger Options Mode, Non-0 Breakpoint Unit**

Defined Identifiers	Description
PREV_UNIT_OR_THIS_TRIGGER	a trigger is generated if the previous breakpoint unit passes in a trigger signal or this breakpoint unit creates a trigger signal
PREV_UNIT_THEN_THIS_TRIGGER	a trigger is generated if the previous breakpoint unit passes in a trigger signal followed by this breakpoint unit creating a trigger signal

## Miscellaneous Trigger Options

The defined identifiers in Table 14.13 are optional.

**Table 14.13 Miscellaneous Trigger Options**

Defined Identifiers	Description
INVERT_B2_COMPARE	the signal from breakpoint 2 is inverted before entering the combination logic; this can be OR'ed into the options parameter of the <b>_eonce_SetTrigger</b> or <b>_eonce_SetCounterTrigger</b> function
EXTEND_COUNTER	the counter, when using the special counting function, is extended to 40-bits by using the OSCNTR as the most significant 24-bits; this can be OR'ed into the options parameter of the <b>_eonce_SetCounterTrigger</b> function when configuring breakpoint unit 0; WARNING: It is not recommended that this option be used if the processor will enter debug mode (breakpoint, console or file I/O) before the counter is read, because the OSCNTR is needed for stepping and would corrupt the counter

## Trace Buffer Capture Options

The options in Table 14.14 determine which kind of changes-of-flow will be captured. OR in as many of the following defined identifiers into the options parameter of the **\_eonce\_SetupTraceBuffer** function.

**Table 14.14 Trace Buffer Capture Options**

Defined Identifiers	Description
CAPTURE_CHANGE_OF_FLOW_NOT_TAKEN	saves target addresses of conditional branches and jumps that are not taken to the trace buffer
CAPTURE_CHANGE_OF_FLOW_INTERRUPT	saves addresses of interrupt vector fetches and target addresses of RTI instructions to the trace buffer
CAPTURE_CHANGE_OF_FLOW_SUBROUTINE	saves the target addresses of JSR, BSR, and RTS instructions to the trace buffer

**Table 14.14 Trace Buffer Capture Options (*continued*)**

Defined Identifiers	Description
CAPTURE_CHANGE_OF_FLOW_0	saves the target addresses of the following taken instructions to the trace buffer: BCC forward branch BRSET forward branch BRCLR forward branch JCC forward and backward branches
CAPTURE_CHANGE_OF_FLOW_1	saves the target addresses of the following taken instructions to the trace buffer: BCC backward branch BRSET backward branch BRCLR backward branch

## Trace Buffer Full Options

The options in Table 14.15 describe what action to take when the trace buffer is full. One of the following defined identifiers must be OR'ed into the options parameter of the **\_eonce\_SetupTraceBuffer** function.

**Table 14.15 Trace Buffer Full Options**

Defined Identifiers	Description
TB_FULL_NO_ACTION	capture continues, overwriting previous entries
TB_FULL_HALT_CAPTURE	capture is halted
TB_FULL_DEBUG	processor enters debug mode
TB_FULL_INTERRUPT	processor interrupts to vector specified as Trace Buffer Interrupt

## Miscellaneous Trace Buffer Option

The TRACE\_BUFFER\_HALTED option may be OR'ed into the options parameter of the **\_eonce\_SetupTraceBuffer** function. This option puts the trace buffer in a halted state when leaving **\_eonce\_SetupTraceBuffer** function. This is most useful when setting a trigger, by calling **\_eonce\_SetTrigger**, to start the trace buffer when a specific condition is met.



# A

## Porting Issues

---

This appendix explains issues relating to successfully porting code to the most current version of the CodeWarrior Development Studio for Freescale 56800/E Hybrid Controllers.

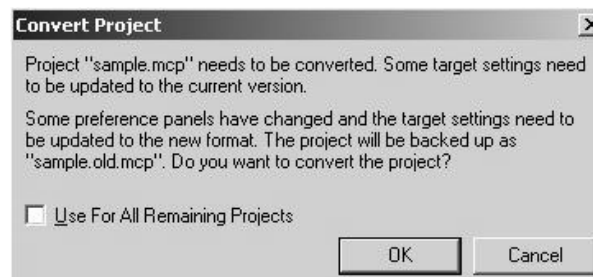
This appendix contains the following sections:

- Converting the DSP56800E Projects from Previous Versions
- Removing "illegal object\_c on pragma directive" Warning

### Converting the DSP56800E Projects from Previous Versions

When you open older projects in the CodeWarrior IDE, the IDE automatically prompts you to convert your existing project (Figure A.1). Your old project will be backed up if you need to access that project file at a later time. The CodeWarrior IDE cannot open older projects if you do not convert them.

**Figure A.1 Project Conversion Dialog**



**Porting Issues**

*Removing "illegal object\_c on pragma directive" Warning*

---

## Removing "illegal object\_c on pragma directive" Warning

If after porting a project to DSP56800E 7.x, you get a warning that says `illegal object_c on pragma directive`, you need to remove it. To remove this warning:

1. Open the project preference and go to the C/C++ Preprocessor.
2. Remove the line `#pragma objective_con` from the prefix text field.

B

# DSP56800x New Project Wizard

This appendix explains the high-level design of the new project wizard.

## Overview

The DSP56800x New Project Wizard supports the DSP56800x processors listed in Table B.1.

Table B.1 Supported DSP56800x Processors for the New Project Wizard

DSP56800	DSP56800E
DSP56F801 (60 MHz)	DSP56852
DSP56F801 (80 MHz)	DSP56853
DSP56F802	DSP56854
DSP56F803	DSP56855
DSP56F805	DSP56857
DSP56F807	DSP56858
DSP56F826	MC56F8322
DSP56F827	MC56F8323
	MC56F8345
	MC56F8346
	MC56F8356
	MC568357
	MC568365
	MC568366

## DSP56800x New Project Wizard Overview

**Table B.1 Supported DSP56800x Processors for the New Project Wizard**

DSP56800	DSP56800E
	MC568367
	MC56F8122
	MC56F8123
	MC56F8145
	MC56F8146
	MC56F8147
	MC56F8155
	MC56F8156
	MC56F8157
	MC56F8165
	MC56F8166
	MC56F8167

Wizard rules for the DSP56800x New Project Wizard are described in the following sub-sections:

- Page Rules
- Resulting Target Rules
- Rule Notes

Click on the following link for details about the DSP56800x New Project Wizard Graphical User Interface:

- [DSP56800x New Project Wizard Graphical User Interface](#)

## Page Rules

The page rules governing the wizard page flow for the simulator and the different processors are shown in the Table B.2, Table B.3, Table B.4, and Table B.5.

**Table B.2 Page Rules for the Simulator, DSP56F801 (60 and 80 MHz) and DSP56F802**

Target Selection Page	Next Page	Next Page
any simulator	Program Choice Page	Finish Page
DSP56F801 60 MHz		
DSP56F801 80 MHz		
DSP56F802		

**Table B.3 Page Rules for the DSP56F803, DSP56F805, DSP56F807, DSP56F826, and DSP56F827**

Target Selection Page	Next Page	Next Page	Next Page
DSP56F803	Program Choice Page	External/Internal Memory Page	Finish Page
DSP56F805			
DSP56807			
DSP56F826			
DSP56F827			

# Freescale Semiconductor, Inc.

## DSP56800x New Project Wizard Overview

---

**Table B.4 Page Rules for the DSP56852, DSP56853, DSP56854, DSP56855, DSP56857, and DSP56858**

Target Selection Page	Next Page	Next Page
DSP56852	Program Choice Page	Finish Page
DSP56853		
DSP56854		
DSP56855		
DSP56857		
DSP56858		

**Table B.5 Page Rules for the MC56F8322, MC56F8323, MC56F8345, MC56F8346, MC56F8356, and MC56F8357**

Target Selection Page	Next Page	Next Page	Next Page if Processor Expert Not Selected	Next Page
MC56F8322	Program Choice Page	Data Memory Model Page	External/Internal Memory Page	Finish Page
MC56F8323				
MC56F8345				
MC56F8346				
MC56F8356				
MC56F8357				
MC56F8365				
MC56F8366				
MC56F8367				
MC56F8122				
MC56F8123				
MC56F8145				
MC56F8146				
MC56F8147				
MC56F8155				
MC56F8156				
MC56F8157				
MC56F8165				
MC56F8166				
MC56F8167				

## Resulting Target Rules

The rules governing possible final project configurations are shown in Table B.6.

**Table B.6 Resulting Target Rules**

Target	Possible Targets
56800 Simulator	Target with Non-HostIO Library and Target with Host IO Library
56800E Simulator	Small Data Model and Large Data Model
DSP5680x	External Memory and/or Internal Memory with pROM-to-xRAM Copy
DSP5682x	External Memory and/or Internal Memory with pROM-to-xRAM Copy
DSP5685x	(Small Data Model and Small Data Model with HSST) or (Large Data Model and Large Data Model with HSST)
MC56F831xx	(Small Data Model and Small Data Model with HSST) or (Large Data Model and Large Data Model with HSST)
MC56F832x	Small Data Model or Large Data Model
MC56F834x	(Small Data Memory External and/or Small Data Memory Internal with pROM-to-xRAM Copy) or (Large Data Memory External and/or Large Data Memory Internal with pROM-to-xRAM Copy)

## Rule Notes

Additional notes for the DSP56800x New Project Wizard rules are:

- The DSP56800x New Project Wizard uses the DSP56800x EABI Stationery for all projects. Anything that is in the DSP56800x EABI Stationery will be in the wizard-created projects depending on the wizard choices.
- The DSP56800x EABI Stationery has all possible targets, streamlined and tuned with the DSP56800x New Project Wizard in mind.
- The DSP56800x New Project Wizard creates the entire simulator project with all the available targets in context of “Stationery as documentation and example.”



## DSP56800x New Project Wizard Graphical User Interface

This section describe the DSP56800x New Project Wizard graphical user interface.

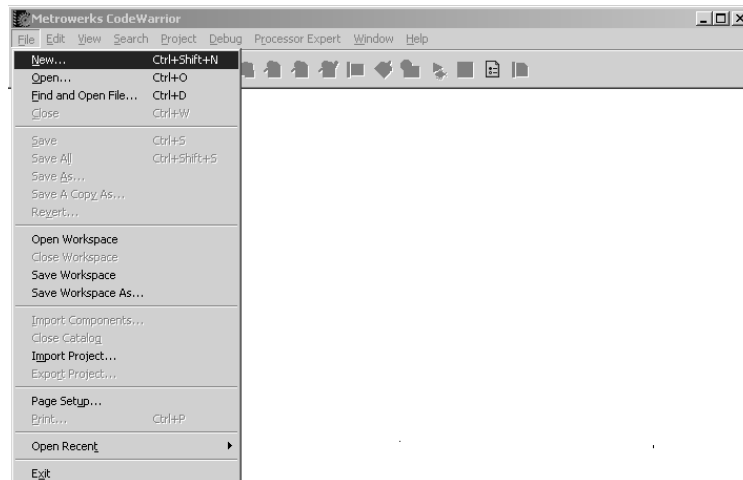
The subsections in this section are:

- Invoking the New Project Wizard
- New Project Dialog Box
- Target Pages
- Program Choice Page
- Data Memory Model Page
- External/Internal Memory Page
- Finish Page

### Invoking the New Project Wizard

To invoke the New Project dialog box, from the Metrowerks CodeWarrior menu bar, select **File>New** (Figure B.1).

**Figure B.1 Invoking the DSP56800x New Project Wizard**



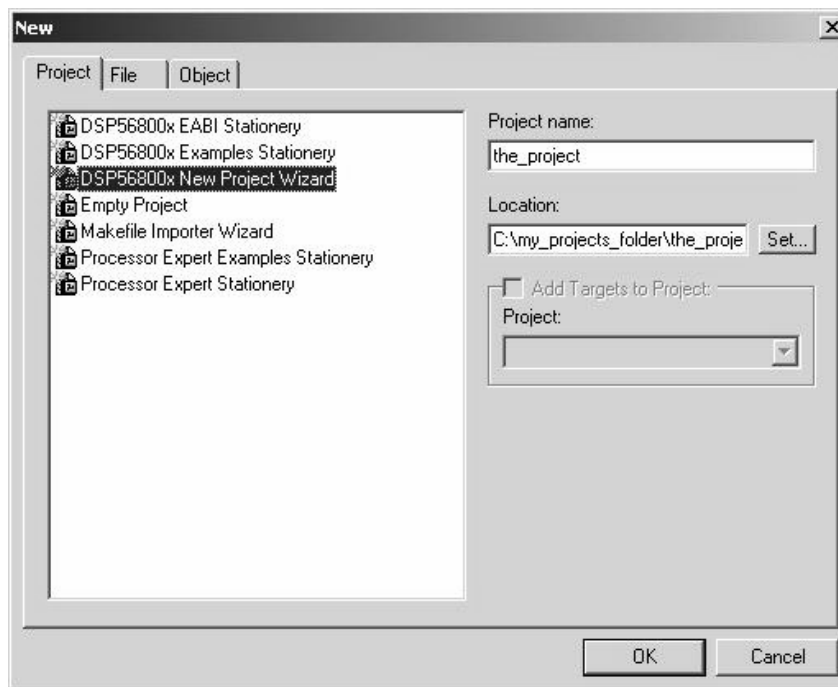
## DSP56800x New Project Wizard

*DSP56800x New Project Wizard Graphical User Interface*

### New Project Dialog Box

After selecting **File>New** from the Metrowerks CodeWarrior menu bar, the New project Dialog Box (Figure B.2) appears. In the list of stationeries, you can select either the “DSP56800x New Project Wizard” or any of the other regular stationery.

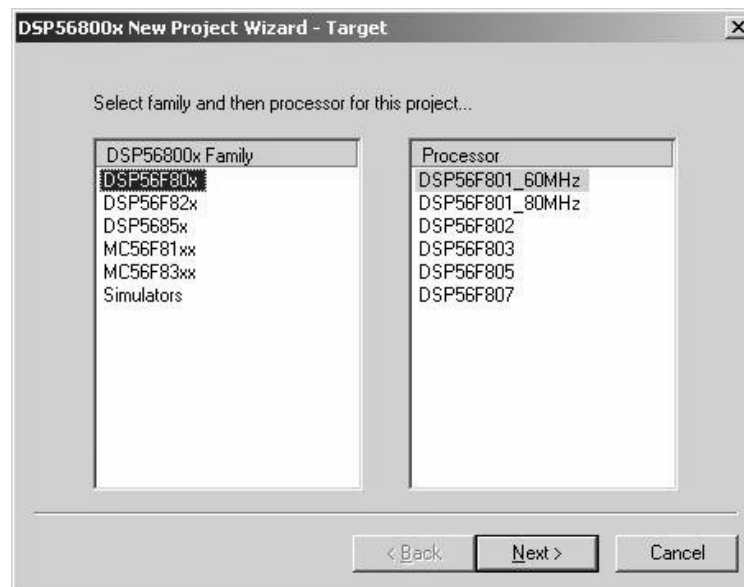
**Figure B.2** New Project Dialog Box



## Target Pages

When invoked, the New Project Wizard first shows a dynamically created list of supported target families and processors or simulators. Each DSP56800x family is associated with a subset of supported processors and a simulator ( Figure B.3, Figure B.4, Figure B.5, Figure B.6, and Figure B.7).

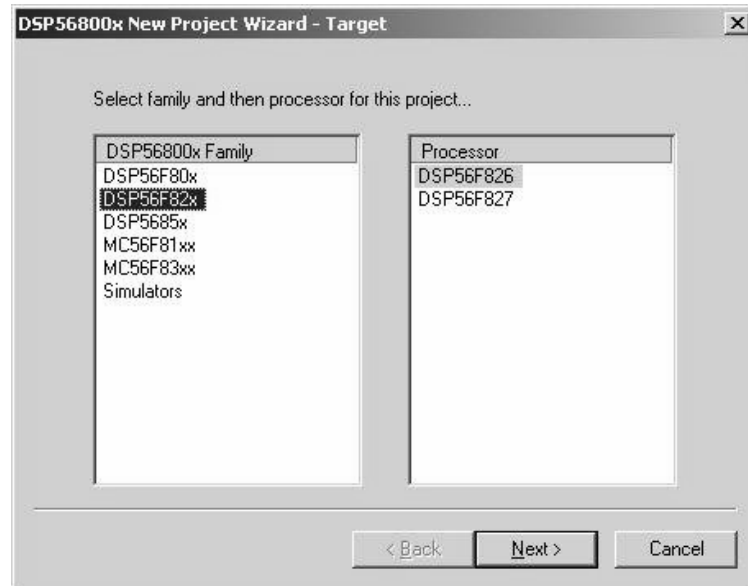
**Figure B.3 DSP56800x New Project Wizard Target Dialog Box (DSP56F80x)**



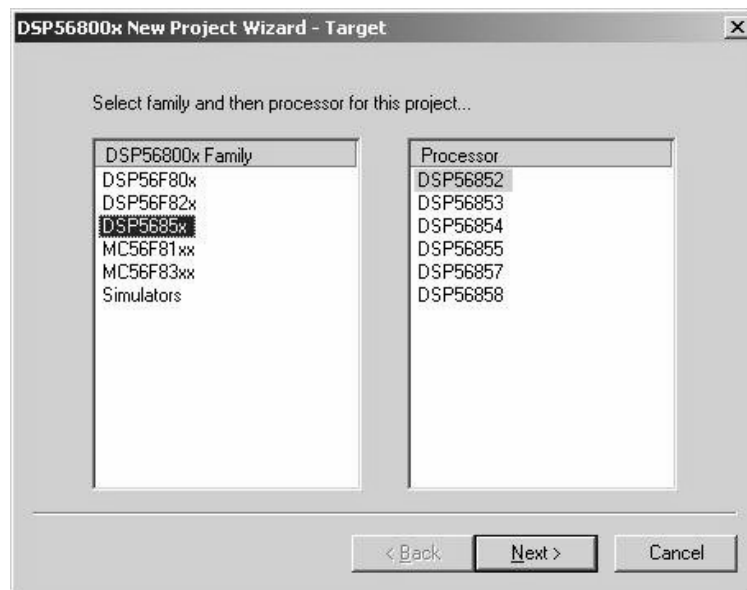
## DSP56800x New Project Wizard

*DSP56800x New Project Wizard Graphical User Interface*

**Figure B.4 DSP56800x New Project Wizard Target Dialog Box (DSP56F82x)**



**Figure B.5 DSP56800x New Project Wizard Target Dialog Box (DSP5685x)**



## DSP56800x New Project Wizard

*DSP56800x New Project Wizard Graphical User Interface*

**Figure B.6 DSP56800x New Project Wizard Target Dialog Box (MC56F83x)**

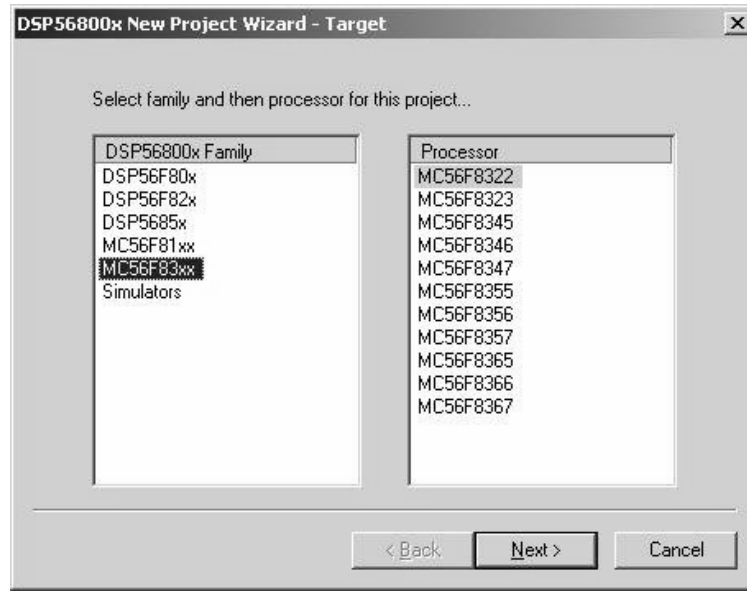
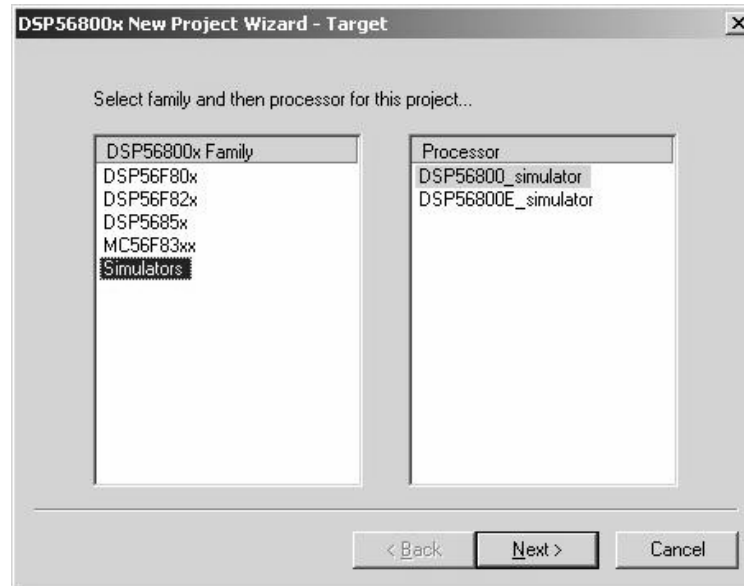


Figure B.7 DSP56800x New Project Wizard Target Dialog Box (Simulator)



One target family and one target processor or simulator must be selected before continuing to the next wizard page.

---

**NOTE**                      Depending on which processor you select, different screens will appear according to the “Page Rules” on page 353.

---

If you choose the simulator, then the DSP56800x New Project Wizard - Program Choice page appears (see “Program Choice Page” on page 363. )

## Program Choice Page

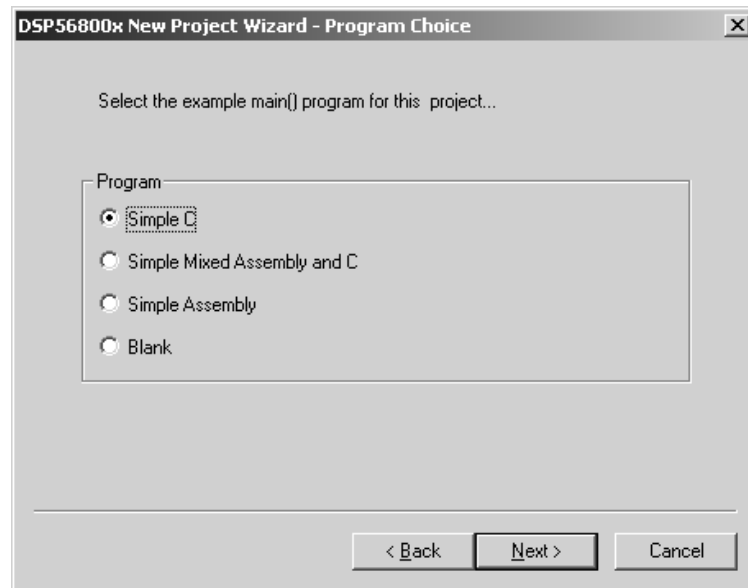
If you chose either of the simulators, then Figure B.8 appears and you can now choose what sort of main() program to include in the project.

## DSP56800x New Project Wizard

*DSP56800x New Project Wizard Graphical User Interface*

---

**Figure B.8 DSP56800x New Project Wizard - Program Choice**



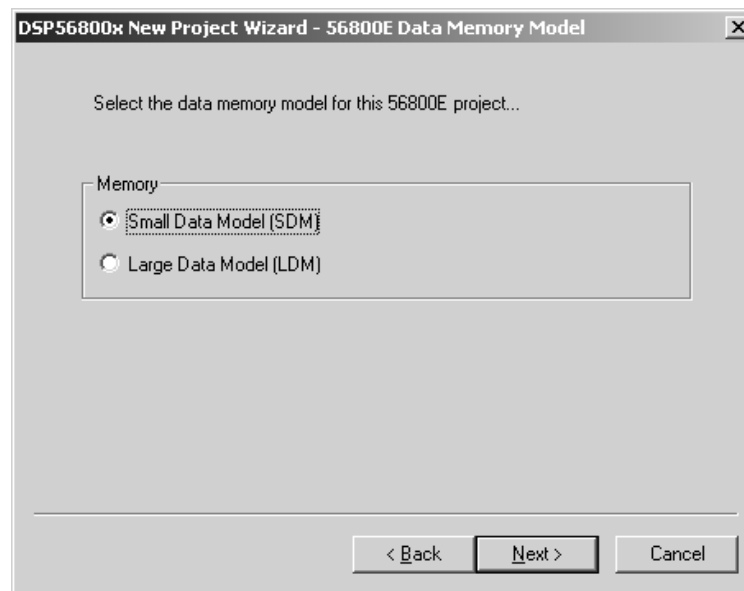
When you click **Next**, the Wizard jumps to the appropriate page determined by the "Page Rules" on page 353.



## Data Memory Model Page

If you select a DSP56800E processor (56F83xx or 5685x family), then the Data Memory Model page appears (Figure B.9) and you must select either the Small Data Model (SDM) or Large Data Model (LDM).

**Figure B.9 DSP56800x New Project Wizard - 56800E Data Memory Model Page**



When you click **Next**, the Wizard jumps to the appropriate page determined by the “Page Rules” on page 353.

## DSP56800x New Project Wizard

DSP56800x New Project Wizard Graphical User Interface

### External/Internal Memory Page

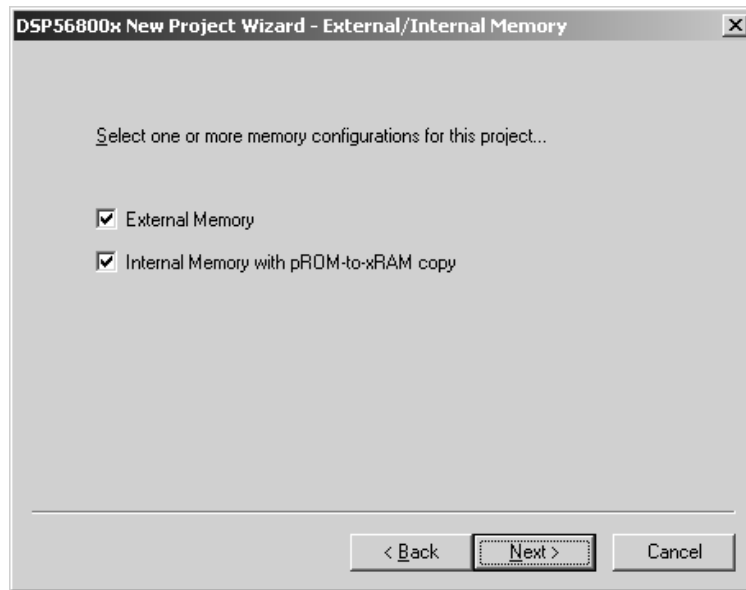
Depending on the processor that you select, the External/Internal Memory page may appear (Figure B.10) and you must select either external or internal memory.

---

**NOTE** Multiple memory targets can be checked.

---

**Figure B.10 DSP56800x New Project Wizard - External/Internal Memory Page**



When you click **Next**, the Wizard jumps to the appropriate page determined by the “Page Rules” on page 353.

## Finish Page

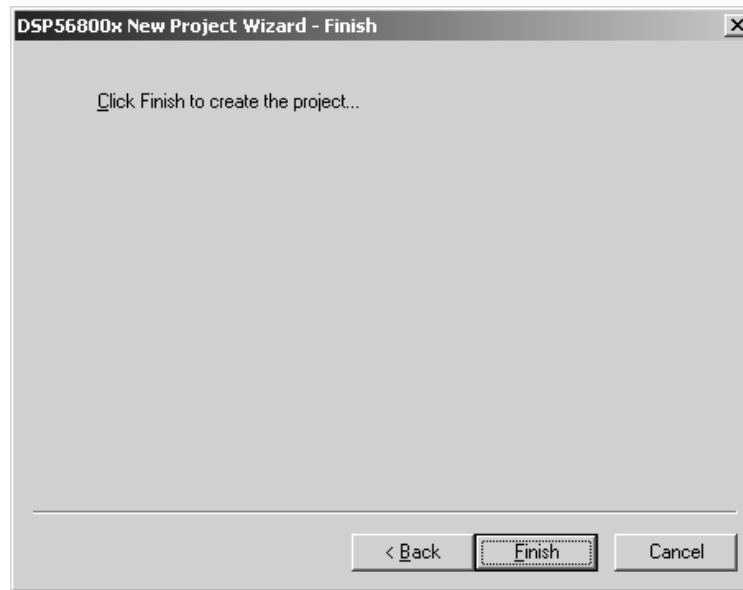
When you click the **Finish** button on the Finish Page (Figure B.11), the project creation process start.

---

**NOTE** All target choices end on this page.

---

**Figure B.11 DSP56800x New Project Wizard - Finish Page**



# Freescale Semiconductor, Inc.

## **DSP56800x New Project Wizard**

*DSP56800x New Project Wizard Graphical User Interface*

---

# Index

---

## Symbols

. (location counter) linker keyword 294, 295  
 .elf file, loading 217  
 \_\_mod\_access intrinsic function 273  
 \_\_mod\_error intrinsic function 275  
 \_\_mod\_getint16 intrinsic function 274  
 \_\_mod\_init intrinsic function 271, 272  
 \_\_mod\_init16 intrinsic function 272  
 \_\_mod\_setint16 intrinsic function 275  
 \_\_mod\_start intrinsic function 272  
 \_\_mod\_stop intrinsic function 273  
 \_\_mod\_update intrinsic function 273  
 \_eonce\_ClearTraceBuffer library function 336, 337  
 \_eonce\_ClearTrigger library function 333  
 \_eonce\_EnableDEBUGEV library function 338  
 \_eonce\_EnableLimitTrigger library function 338, 339  
 \_eonce\_GetCounters library function 334  
 \_eonce\_GetCounterStatus library function 334, 335  
 \_eonce\_GetTraceBuffer library function 336  
 \_eonce\_HaltTraceBuffer library function 337, 338  
 \_eonce\_Initialize library function 330  
 \_eonce\_SetCounterTrigger library function 332, 333  
 \_eonce\_SetTrigger library function 331, 332  
 \_eonce\_SetupTraceBuffer library function 335  
 \_eonce\_StartTraceBuffer library function 337

## A

abs\_s intrinsic function 237  
 Access Paths panel 48  
 add intrinsic function 239, 240  
 add\_hfm\_unit flash debugger command 221  
 ADDR linker keyword 295, 296  
 ALIGN linker keyword 296  
 ALIGNALL linker keyword 296, 297  
 Auto-clear previous breakpoint on new breakpoint release 73

## B

bean inspector window 85, 90, 91  
 bean selector window 84, 89–90  
 breakpoints 192, 193  
 Build Extras panel 48

## C

C for DSP56800E 119–146  
 C/C++ language panel 51  
 C/C++ warnings panel 56–59  
 calling conventions 121–124  
 Changing Target Settings 45  
 child windows 32, 33  
 code storage 139  
 CodeWarrior IDE 13, 14, 35, 36  
     installing 23  
     installing and registering 20  
 CodeWarrior IDE Target Settings Panels 48  
 command converter server 180–188  
 command window 218  
 conventions, calling 121–124  
 converting CodeWarrior projects 349  
 CPU types overview window 98  
 creating a project 29, 34  
 Custom Keywords settings panel 48  
 Cycle/Instruction Count 215

## D

data alignment 131, 133  
 data storage 139  
 deadstripping 145  
 debugger  
     command converter server 180–188  
     EOnCE features 206–214  
     fill memory 202–204  
     load/save memory 200–202  
     operating 188–195  
     save/restore registers 204–206  
     system level connect 219  
 Debugger Settings panel 48  
 debugging 179–224  
     flash memory 219  
     notes for hardware 223  
     target settings 179, 180  
 development process 36–42  
     building (compiling and linking) 40–42  
     debugging 42  
     editing code 39, 40  
     project files 38, 39

---

development studio overview 35–42  
dialog boxes  
    fill memory 202–204  
    load/save memory 200–202  
    save/restore registers 204–206  
directories, installation 23  
div\_ls intrinsic function 248, 249  
div\_ls4q intrinsic function 249  
div\_s intrinsic function 247, 248  
div\_s4q intrinsic function 248  
docking windows 32, 33  
DSP56800E simulator 214

## E

ELF disassembler panel 65–67  
EOnCE debugger features 206–214  
EOnCE library  
    definitions 339–348  
EOnCE library functions 328–339  
    \_eonce\_ClearTraceBuffer 336, 337  
    \_eonce\_ClearTrigger 333  
    \_eonce\_EnableDEBUGEV 338  
    \_eonce\_EnableLimitTrigger 338, 339  
    \_eonce\_GetCounters 334  
    \_eonce\_GetCounterStatus 334, 335  
    \_eonce\_GetTraceBuffer 336  
    \_eonce\_HaltTraceBuffer 337, 338  
    \_eonce\_Initialize 330  
    \_eonce\_SetCounterTrigger 332, 333  
    \_eonce\_SetTrigger 331, 332  
    \_eonce\_SetupTraceBuffer 335  
    \_eonce\_StartTraceBuffer 337  
EOnCE panels  
    set hardware breakpoint 207, 208  
    set trigger 212–214  
    special counters 208–209  
    trace buffer 209–212  
example HSST host program 160–162  
example HSST target program 169, 170  
Exporting and importing panel options to XML Files 47  
extract\_h intrinsic function 245  
extract\_l intrinsic function 245, 246

## F

ffs\_l intrinsic function 257, 258  
ffs\_s intrinsic function 256  
File Mappings panel 48

fill memory dialog box 202–204  
flash debugger commands  
    add\_hfm\_unit 221  
    set\_hfm\_base 220  
    set\_hfm\_config\_base 221  
    set\_hfm\_erase\_mode 221  
    set\_hfm\_verify\_erase 222  
    set\_hfm\_verify\_program 222  
    set\_hfmclkd 220  
    target\_code\_sets\_hfmclkd 222  
flash memory debugging 219  
Flash ROM  
    programming tips 223  
floating windows 32, 33  
FORCE\_ACTIVE linker keyword 297  
formats, number 120, 121

## G

getting started 19, 29, 34  
Global Optimizations settings panel 48

## H

hardware debugging notes 223  
high-speed simultaneous transfer 153–170  
host program example, HSST 160–162  
host-side API hsst functions 153–160  
HSST 153–170  
    host-side API functions 153–160  
    target library API functions 162–169  
    visualization 171  
HSST functions  
    hsst\_attach\_listener 158, 159  
    hsst\_block\_mode 157, 158  
    HSST\_close 162, 163  
    hsst\_close 154  
    hsst\_detach\_listener 159  
    HSST\_flush 166  
    hsst\_noblock\_mode 158  
    HSST\_open 162  
    hsst\_open 153  
    HSST\_raw\_read 167  
    HSST\_raw\_write 167, 168  
    HSST\_read 165  
    hsst\_read 155  
    HSST\_set\_log\_dir 168, 169  
    hsst\_set\_log\_dir 160  
    HSST\_setvbuf 163, 164

---

HSST_size 166	__mod_init 271, 272
hsst_size 157	__mod_init16 272
HSST_write 164, 165	__mod_setint16 275
hsst_write 156	__mod_start 272
HSST host program example 160–162	__mod_stop 273
HSST target program example 169, 170	__mod_update 273
hsst_attach_listener function 158, 159	abs_s 237
hsst_block_mode function 157, 158	add 239, 240
HSST_close function 162, 163	div_ls 248, 249
hsst_close function 154	div_ls4q 249
hsst_detach_listener function 159	div_s 247, 248
HSST_flush function 166	div_s4q 248
hsst_noblock_mode function 158	extract_h 245
HSST_open function 162	extract_l 245, 246
hsst_open function 153	ffs_l 257, 258
HSST_raw_read function 167	ffs_s 256
HSST_raw_write function 167, 168	fractional arithmetic 234, 235
HSST_read function 165	implementation 233, 234
hsst_read function 155	L_abs 238
HSST_set_log_dir function 168, 169	L_add 241
hsst_set_log_dir function 160	L_deposit_h 246
HSST_setvbuf function 163, 164	L_deposit_l 246, 247
HSST_size function 166	L_mac 253
hsst_size function 157	L_msu 254
HSST_write function 164, 165	L_mult 254, 255
hsst_write function 156	L_mult_ls 255
	L_negate 239
	L_shl 265, 266
	L_shlftNs 266
	L_shlfts 267
	L_shr 267, 268
	L_shr_r 268, 269
	L_shrtNs 269
	L_sub 241, 242
	mac_r 250, 251
	msu_r 251
	mult 252
	mult_r 252, 253
	negate 237, 238
	norm_l 258
	norm_s 256, 257
	round 259
	shl 260, 261
	shlftNs 261, 262
	shlfts 262, 263
	shr 263
	shr_r 264
	shrtNs 264, 265
	stop 242

---

---

- sub 240
- turn\_off\_conv\_rndg 243
- turn\_off\_sat 243, 244
- turn\_on\_conv\_rndg 244
- wait 243

introduction 13–17

## K

KEEP\_SECTION linker keyword 298

## L

- L\_abs intrinsic function 238
- L\_add intrinsic function 241
- L\_deposit\_h intrinsic function 246
- L\_deposit\_l intrinsic function 246, 247
- L\_mac intrinsic function 253
- L\_msu intrinsic function 254
- L\_mult intrinsic function 254, 255
- L\_mult\_ls intrinsic function 255
- L\_negate intrinsic function 239
- L\_shl intrinsic function 265, 266
- L\_shlftNs intrinsic function 266
- L\_shlfts intrinsic function 267
- L\_shr intrinsic function 267, 268
- L\_shr\_r intrinsic function 268, 269
- L\_shrtNs intrinsic function 269
- L\_sub intrinsic function 241, 242
- large data model support 141–144
- libraries and runtime code 321–348
- link order 145
- linker command files
  - keywords 294–304
  - structure 281–284
  - syntax 284–294
- linker keywords
  - . (location counter) 294, 295
  - ADDR 295, 296
  - ALIGN 296
  - ALIGNALL 296, 297
  - FORCE\_ACTIVE 297
  - INCLUDE 297
  - KEEP\_SECTION 298
  - MEMORY 298, 299
  - OBJECT 300
  - REF\_INCLUDE 300
  - SECTIONS 300, 302

- SZEOF 302
- SZEOFW 303
- WRITEB 303
- WRITEH 303
- WRITEW 304

load/save memory dialog box 200–202

loading .elf file 217

## M

- M5600E target panel 50, 51
- M56800E assembler panel 60, 62
- M56800E linker panel 67–71
- M56800E processor panel 62
- M56800E target (debugging) panel 73–77
- mac\_r intrinsic function 250, 251
- math support intrinsic functions 235–269
- MEMORY linker keyword 298, 299
- memory map window 97, 98
- memory, viewing 195–199
- Metrowerks Standard Library (MSL) 321–325
- modulo addressing
  - error codes 278, 279
  - intrinsic functions 269–279
  - points to remember 277, 278
- modulo buffer examples 275–277
- msu\_r intrinsic function 251
- mult intrinsic function 252
- mult\_r intrinsic function 252, 253

## N

- negate intrinsic function 237, 238
- norm\_l intrinsic function 258
- norm\_s intrinsic function 256, 257
- number formats 120, 121

## O

- OBJECT linker keyword 300
- operating the debugger 188–195
- optimizing code 144, 145
- overview, development studio 35–42
- overview, target settings 45

## P

- P memory, viewing 197–199
- panels



- 
- C/C++ language 51
  - C/C++ warnings 56–59
  - ELF disassembler 65–67
  - M56800E assembler 60, 62
  - M56800E linker 67–71
  - M56800E processor 62
  - M56800E target 50, 51
  - M56800E target (debugging) 73–77
  - remote debug options 77–79
  - remote debugging 72–73
  - target settings 49–50
  - panels, settings 48–79
  - Peripheral Module Registers 146
  - peripherals usage inspector window 101
  - porting issues 349
  - Processor Expert
    - beans 83–85
    - code generation 82–83
    - menu 85–88
    - overview 81–88
    - page 83
    - tutorial 102–118
  - Processor Expert interface 81–118
  - Processor Expert windows 89–101
    - bean inspector 90, 91
    - bean selector 89–90
    - CPU types overview 98
    - installed beans overview 100
    - memory map 97, 98
    - peripherals usage inspector 101
    - resource meter 99
    - target CPU 92–96
  - project
    - creating 29, 34
- R**
- REF\_INCLUDE linker keyword 300
  - references 17
  - register details window 199, 216
  - register values 193–195
  - Registers, peripheral module 146
  - remote debug options panel 77–79
  - remote debugging panel 72–73
  - requirements, system 19
  - resource meter window 99
  - Restoring Target Settings 47
  - round intrinsic function 259
  - runtime code 321–348
  - runtime initialization 325–328
- S**
- save/restore registers dialog box 204–206
  - Saving new target settings
    - stationery files 47
  - SECTIONS linker keyword 300, 302
  - set hardware breakpoint EOnCE panel 207, 208
  - set trigger EOnCE panel 212–214
  - set\_hflkd flash debugger command 220
  - set\_hfm\_base flash debugger command 220
  - set\_hfm\_config\_base flash debugger command 221
  - set\_hfm\_erase\_mode flash debugger command 221
  - set\_hfm\_verify\_erase flash debugger command 222
  - set\_hfm\_verify\_program flash debugger command 222
  - settings panels 48–79
    - Access Paths 48
    - Build Extras 48
    - C/C++ language 51
    - C/C++ warnings 56–59
    - Custom Keywords 48
    - Debugger Settings 48
    - ELF disassembler 65–67
    - File Mappings 48
    - Global Optimizations 48
    - M56800E assembler 60, 62
    - M56800E linker 67–71
    - M56800E processor 62
    - M56800E target 50, 51
    - M56800E target (debugging) 73–77
    - remote debug options 77–79
    - remote debugging 72–73
    - Source Trees 48
    - target settings 49–50
  - settings, target 43–79
  - shl intrinsic function 260, 261
  - shlftNs intrinsic function 261, 262
  - shlfts intrinsic function 262, 263
  - shr intrinsic function 263
  - shr\_r intrinsic function 264
  - shrtNs intrinsic function 264, 265
  - simulator 214
  - simultaneous transfer, high speed 153–170
  - SIZEOF linker keyword 302
  - SIZEOFW linker keyword 303
  - Source Trees settings panel 48

---

special counters EOnCE panel 208–209  
stack frames 125, 126  
stationery  
    saving new target settings 47  
stop intrinsic function 242  
storage, code and data 139  
sub intrinsic function 240  
system level connect 219  
system requirements 19

## T

target CPU window 92–96  
target library API hsst functions 162–169  
target program example, HSST 169, 170  
target settings 43–79  
    overview 45  
target settings panel 49–50  
Target Settings panels  
    Access Paths 48  
    Build Extras 48  
    Custom Keywords 48  
    Debugger Settings 48  
    File Mappings 48  
    Global Optimizations 48  
    Source Trees 48  
Target Settings window 46  
target\_code\_sets\_hfmcld flash debugger  
    command 222  
trace buffer EOnCE panel 209–212  
turn\_off\_conv\_rndg intrinsic function 243  
turn\_off\_sat intrinsic function 243, 244  
turn\_on\_conv\_rndg intrinsic function 244  
tutorial, Processor Expert 102–118

## U

undocking windows 32, 33

## V

values, register 193–195  
viewing memory 195–199

## W

wait intrinsic function 243  
windows  
    bean inspector 85, 90, 91

bean selector 84, 89–90  
CPU types overview 98  
installed beans overview 100  
memory map 97, 98  
peripherals usage inspector 101  
Processor Expert 89–101  
    register details 199, 216  
    resource meter 99  
    target CPU 92–96  
WRITEB linker keyword 303  
WRITEH linker keyword 303  
WRITEW linker keyword 304

## X

X memory, viewing 195–197  
XML files  
    exporting and importing panel options 47